UNIVERSITY OF CAMBRIDGE

Software-based approximate computing for mathematical functions

Nicholas Gerard Timmons

Downing College

This dissertation is submitted on April, 2021 for the degree of Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Nicholas Gerard Timmons
April, 2021

Abstract

Software-based approximate computing for mathematical functions
Nicholas Gerard Timmons

The four arithmetic floating-point operations ($+$, $-$, $\div$ and $\times$) have been precisely specified in IEEE-754 since 1985, but the situation for floating-point mathematical libraries and even some hardware operations such as *fused multiply-add* is more nuanced as there are varying opinions on which standards should be followed and when it is acceptable to allow some error or when it is necessary to be correctly-rounded.

Deterministic correctly-rounded elementary mathematical functions are important in many applications. Others are tolerant to some level of error and would benefit from less accurate, better-performing approximations. We found that, despite IEEE-754 (2008 and 2019 only) specifying that 'recommended functions' such as *sin*, *cos* or *log* should be correctly rounded, the mathematical libraries available through standard interfaces in popular programming languages provide neither correct-rounding nor maximally performing approximations, partly due to the differing accuracy requirements of these functions in conflicting standards provided for some languages, such as C. This dissertation seeks to explore the current methods used for the implementation of mathematical functions, show the error present in them and demonstrate methods to produce both low-cost correctly-rounded solutions and better approximations for specific use-cases.

This is achieved by:

First, exploring the error within existing mathematical libraries and examining how it is impacting existing applications and the development of programming language standards.

We then make two contributions which address the accuracy and standard conformance problems that were found: 1) an approach for a correctly-rounded 32-bit implementation of the elementary functions with minimal additional performance cost on modern hardware; and 2) an approach for developing a better performing incorrectly-rounded solution for use when some error is acceptable and conforming with the IEEE-754 standard is not a requirement. For the latter contribution, we introduce a tool for semi-automated generic code sensitivity analysis and approximation.

Next, we target the creation of approximations for the standard activation functions used in neural networks. Identifying that significant time is spent in the computation of the activation functions, we generate approximations with different levels of error and better performance characteristics. These functions are then tested in standard neural networks to determine if the approximations have any detrimental effect on the output of the network. We show that, for many networks and activation functions, very coarse approximations are suitable replacements to train the networks equally well at a lower overall time cost.

This dissertation makes original contributions to the area of approximate computing. We demonstrate new approaches to safe-approximation and justify approximate computation generally by showing that existing mathematical libraries are already suffering the downsides of approximation and latent error without fully exploiting the optimisation space available due to the existing tolerance to that error and showing that correctly-rounded solutions are possible without a significant performance impact for many 32-bit mathematical functions.

Acknowledgements

This thesis was written during a time in my life, and the world, of both great upheaval and a lot of isolation. Without the selfless support of people who themselves were going through periods of great and often quiet difficulty I would not have been able to reach this goal. I would like to thank those people in my life, with a special mention to my supervisor, Andy Rice, and my advisor, Alan Mycroft, who have been consistently consistent throughout this process even while I have been erratically erratic.

I would also like to mention the members of the Cambridge University American Football Club. While the varying performance of our team on the field has been full of highs and lows, off the field the support and camaraderie of my teammates has been an endless source of positively and the purest enthusiasm through every season.

Contents

Chapter 1

Introduction

When computers were first becoming popular tools, one of their first uses was scientific computing. There was no standard for representing real numbers or how to compute elementary functions correctly in these new numerical systems. This caused many libraries to be written with different standards of error and with different approaches to representing real numbers. As a result, it was fairly standard to use arbitrary approximations which produced results good enough for the task to be used.

Approximations were a necessity for the computation of some mathematical problems in a reasonable time. The need for very high precision for most tasks did not exist, and many computers were specialised in those tasks for which they were built. As computers became more mature and code began to be shared between computing devices, standards started to develop around how programs should handle numbers and numerical functions. This culminated in, what is for this dissertation at least, one of the most important and still developing standards for numerical computing, the *IEEE-754 Standard for floating-point arithmetic* (IEEE-754). This defines how floating-point numbers should be represented and handled and it also specifies important mathematical functions used in scientific computing.

Today we have settled on floating-point as the numerical system for representing real numbers in most scenarios. This is defined as part of the still-evolving IEEE-754. The mathematical functions used in scientific computing are also defined in this standard. Importantly for our research into approximate computation, the standard requires the mathematical functions to be correctly rounded — they should return the closest representable value to the true value for the given rounding mode.

We know that many applications require results that are precise, deterministic and portable, while other applications are not only tolerant to error but are infeasible to be run without it. A motivation for this work was to provide methods for the use of safe approximations alongside the correctly-rounded implementation. Programmers should have the ability to specify the level of error which is acceptable to them beyond the constraints of simply switching between 32- and 64-bit data types so that code can be written more efficiently and save on operations that calculate an answer beyond the precision needed. For any given program, there is also a level of error in the final output that will be accepted. For some programs it is zero, for others it is a lot, and some engineers may want to trade some level of accuracy for a gain in performance.

-In many ways, CPU performance gains are less and less each year; the so-called 'Free Lunch'[1] is over. Where we could once rely on increasing performance to reach new heights of processing, it is now again necessary to look to where we can cut corners, not waste calculations and get the answers just a little bit wrong in ways which may not matter so that we can reach our performance goals, just as the first computing engineers had to do. This means better optimisation and not wasting performance doing things that do not need to be done. When approximate computation is mentioned, the initial thoughts may be about getting the answer close enough, or not as correct as it could be. A better way to think about it that is more representative of the work currently in the field, is doing close to the optimal amount of work for the task. When you use an approximation in place of a high-precision result and it does not negatively affect the outcome from the program, you have not calculated the wrong answer. Rather, you have optimised away many redundant calculations. The true optimal solution for a program can achieve the desired output with the minimum amount of work.

This motivated my research into approximate computing for elementary mathematical functions and the accuracy of existing popular mathematical libraries. I hoped to find approaches suitable for an engineer when it would be possible to switch to an approximation for additional performance when needed.

A surprising finding when I began investigating the implementation of elementary functions in popular mathematical libraries was that they were already approximations. Not in the sense that they approximate to the level of precision of the floating-point data type we are using, but that they were implemented to not produce the correctly-rounded answer at their given precision. I learned that this was by design, for performance reasons — the exact goal I had set to provide to engineers, but only *when it was safe to do so*.

This was unexpected because the IEEE-754 standard explicitly states that these functions should be correctly-rounded. In not following this aspect of the standard, the libraries' functions will produce different results for the same inputs for a significant range of their input domain. This means that if one computer links to one library and another links to a different library, despite the standard guaranteeing that they should get the same results, they might not. In part this is due to not all languages implementing this standard, by design, choosing not to follow the correct-rounding guarantee of the standard.

A major part of the work in this dissertation is revealing the existing level of accepted (knowingly or otherwise) error that is present but mostly undocumented in common standard mathematical libraries and then showing the negative effects of this on programs that rely on

[1] A term referring to a programmer not having to overly worry about the performance of the program they were writing, because by the time they spent any significant effort optimising it then there would already be a faster CPU on the market that made the gains insignificant.

those functions being accurate. In doing so, I show that there is already an accepted level of error for these functions in some uses and I argue it should be exploited for better run-time performance.

While finding that the standard implementations of the mathematical functions were inaccurate was surprising, it shows that even in modern real-world computing, errors can be accepted. The reasons given for this by developers who work on these libraries was that the performance cost of correct-rounding in these functions was not worth the small amount of additional accuracy. This is particularly important. It shows that real developers are not only willing to sacrifice accuracy for performance gain, but are already doing so in key functions. This draws us towards an interesting question, which this dissertation attempts to explore and provide some answers to: If a small amount of error went almost unnoticed for a long time in widely used functions, then where is the true error threshold, and how much error is acceptable?

In this dissertation, I analyse the existing error which is tolerated in the elementary functions of standard mathematical libraries. In doing so I make two arguments.

First, I argue that we do need the correct answer sometimes. Correctly-rounded solutions should be available because it is in the standard and expected. I present a method for achieving this cheaply on modern hardware. If we do not have correct answers in the standard implementations, it makes it extremely hard to argue for approximations because the current solutions (which many people believe to be correct) are already running as fast as an approximation, because they are an approximation. I back up our claims for the need for correct-rounding by showing the existing issues caused by the inconsistencies between implementations. In particular, I show how it affects the portability and determinism of some algorithms and programs.

Second, I argue that sometimes the exact correct answer is not needed. I present a better approximation for the same error-bounds present in the current implementations of an elementary function, and in doing so show a method for finding safe approximations with tools I have developed. I then demonstrate that the activation functions within a neural network are very error-tolerant and that by taking advantage of the acceptability of that error, significantly reduced training times can be achieved when compared to the accurate solutions without any penalty to the result.

I believe that this makes a significant contribution to the current understanding of approximation for numerical computing. I reveal the current state of play, show how to fix it and how to move forward if you do want to use approximations to gain better performance in the mathematical software which uses them.

1.1 Motivation

In Hyde's book, *Write Great Code* [82] he states:

> *As computer system performance has increased from megahertz to hundreds of megahertz to gigahertz, computer software performance has taken a back seat to other concerns. Today, it's not at all uncommon for software engineers to exclaim, "You should never optimise your code!" Funny, you don't hear too many software users making such statements.*

Optimisation is important to making good software. Code that is not optimal is uneconomical. When code takes a long time to run, fewer experiments or programs can be run at the same time. Slow code wastes time.[2] It also makes for steeper hardware requirements to perform tasks, which means spending more money on cloud computing time or in-house hardware.

Non-optimal code externalises the costs from the engineer of the code to the user in terms of time, hardware and energy. This work was motivated by a desire to find techniques and develop tools that can help to reduce this cost by providing new approaches to approximation that can be easily adopted when writing or maintaining mathematical code. In the search for these techniques, I found that many mathematical libraries which were expected to be accurate were producing small errors. This motivated us to find out where the error in these libraries was coming from and how tolerated it is and bring that together with our existing research into sensitivity analysis and approximations.

This gave a target of widely used elementary mathematical functions which were already resilient to some level of error to investigate further optimisation, both in isolation and within larger applications. This helps to reduce the larger problem of approximations for mathematical software down to a small number of functions that have well-defined standards and a long history of how they are used and approximated.[3]

With this more limited scope, there is a task *approximate functions to the optimal level of work*, a target *the elementary functions provided by standard mathematical libraries* and a metric for success *the programs being faster and not causing detrimental output.* This is the direction from which I have approached this work. It is direct, with a lot of scope for investigating how different approaches to output in different uses result in varying boundaries and classifications for the acceptability of the output.

1.1.1 Approach to Approximation

There is a reason that approximations are not widely applied when it comes to mathematical computing: *good approximations are hard.*

Making an approximation is easy. A stopped clock is an approximation of a working clock, it is correct twice a day. But clearly, that approximation is unhelpful. Approximations must

[2] See *xkcd* 303. https://xkcd.com/303/

[3] And by long history, I mean the oldest approximation investigated were ancient Babylonian approximations of $\sqrt{2}$ to approximately six base-10 digits that were found on a clay tablet (*YBC 7289*) from roughly 1600 years BCE [54].

first give a benefit to the application in some way (usually run-time performance or memory use), they must gain this benefit without adversely hindering the results beyond the acceptable error for all valid inputs, and they must be stable. If the approximation is unable to meet these requirements, it is unlikely to be adopted.

The next difficulty is that the approximation must advertise its flaws. It would be easy to create a whole library of fast mathematical functions that were ten times the speed of the standard implementations, but if you do not tell the users that they are only accurate to 10% of the true answer for that function the user will use them in places that require more accuracy and again you have a problem. Approximations need well-defined conditions for their creation and well-advertised descriptions of how they can be used. Without this, it is natural for a developer to not trust approximations. Any developer would be frustrated if they ran into bugs in a library because it did not behave how they expected it to behave.

In the past[4], approximations have been used either implicitly without informing the user, or explicitly with no guarantees of actual behaviour. This has led to distrust when people encounter them. Consider your feelings if you saw a check-in to your favourite repository with the message '5x run-time improvement, only got the answer wrong a little'. You would likely be somewhat concerned.

If we are approaching a time where the rate of the increase of performance in computing is slowing and there is a practical and ethical need to more efficiently use the available computing power, then it is time for approximations to be made public, easy to understand and easy to safely apply. I want to remove the stigma by providing tools, methodologies and proof of the effectiveness of appropriate and safe approximations. Without a solid basis for approximations to be applied, they are just messy hacks and will generally not be used.

Some people have approached approximation as an analytical, mathematical problem to solve, using formal proofs or interval arithmetic to provide well-defined solutions for well-defined problems. For some areas of computing, this is possible and incredibly useful. However, I am more interested in approaches that treat source-code generically without the formal bounds you would find in proofs. Approaching approximation more empirically because so much code used in the real world is ill-defined and the domains, rules and use-cases where it is executed can change as a project develops. For this reason, I am interested in providing tools and approaches that would work with less well-defined scenarios so that they can be adapted for use in modern real-world projects looking for improved performance.

1.2 Structure

I begin in Chapter 2 with a review of existing approaches to approximations in modern computing and a short discussion of how our work fits within that context.

[4] Generally, with a few notable exceptions.

Chapter 3 goes into further detail on the more relevant background topics needed to understand the work in this dissertation by looking at the historical background of mathematical computing, numerical standards, approaches to approximation and some small examples of the different types of approximations that have been used and how they require different understandings of acceptable error. This background also covers the current views of equality for functions and data in a computing environment so that we have grounding when suggesting it is acceptable to replace a function with a close approximation.

The differences in how I consider equality is pivotal when considering approximation. *If refactoring a function results in minor changes to the floating-point behaviour, are the two versions still equal?* A compiler would say no (except under compilation with explicit flags to allow floating-point error[5]) and not allow this transform, whereas it is something programmers routinely change in day-to-day code maintenance. Effectively defining the different types of equality in software and providing a language for it allows the later chapters to explain more precisely what I mean by replacing a function with an 'equal' approximation.

The next chapters begin our contributions to the field, starting in Chapter 4 with an analysis of existing error in the elementary mathematical functions used in common mathematical libraries which ship with popular operating systems[6]. I demonstrate the existing problems with the undocumented errors in these libraries by providing a few examples of simple programs, including neural networks, which are not portable or which produce incorrect results because of the approximation error and the different position of the errors between implementations. I show how the error is inconsistent and undocumented between versions of the same library, causing security and portability risks. The mathematical libraries are compared to similar GPU implementations to show that the problem only gets worse when moved to a device more devoted to competing on overall performance.

In Chapter 5, to show that some of the problems caused by incorrect-rounding can be avoided without excessive cost I provide a correctly-rounded 32-bit solution for the mathematical functions which has better or equal performance to the existing 64-bit implementations. I show that these new implementations are not much more expensive than the current incorrectly-rounded solutions and argue for them to be included in place of the existing solution.

Now that correctly rounded solutions have been provided for where they are needed, in Chapter 6 I provide a better performing one-ULP[7] max-error solution for the `sin` function to be used in current applications which are tolerant of the one-ULP max-error present in existing mathematical libraries. I use the ApproximateComputations.jl library that I have developed to analyse the accuracy and performance of the Julia implementation of the function and show

[5] For example, GCC, Clang and MSVC all provide compiler flags to allow optimisations which allow some error in floating-point calculations. This is the '*-fast-math*' flags for GCC and Clang, and '*/fp:fast*' for MSVC

[6] You might be surprised to know there is error in these libraries; it is not well advertised or documented.

[7] ULP is Unit-in-the-Last-Place. A measure of the distance between one floating-point value and the next based on the smallest representative value for a given floating-point value. Full explanation in Section 3.1.3.

how to improve the implementation when aiming for the actual worst-case error they already exhibit, instead of aiming for correct-rounding. I then show how a more optimal function can be generated where the average ULP error is higher but never exceeds the one-ULP max-error limit.

Finally, to show this work in a real-world context. In Chapter 7, approximations of varying behaviour and accuracy of the standard activation functions used in neural networks are produced and used in place of the standard implementations. Through a number of benchmarked networks, I show that coarse approximations with bounded behaviours are safe to use and produce equally well-trained networks with a significant reduction in the overall training time. I then show that using the same functions for inference will improve the run-time performance of many neural networks.

Chapter 2

Related Work

Approximate computation approaches are diverse and varied. Many focus upon the use of domain-specific techniques which are designed for implementation within highly specified contexts. These techniques are not flexible, and although valuable for research applications, they are in many cases unsuitable for generalised use and few attempts have been made to resolve the challenges that limit the applicability of approximate computation techniques to mainstream software development [113].

In modern software development, source-code is written and then processed in many different forms before it becomes an executable. From the source-code to the compilation process and final execution there are many stages where it is possible to introduce approximations. Each stage presents its own constraints and limitations. Some of these stages, such as hardware, are highly specific. If these stages are targeted as the main point of approximation, the ability of the technique to reach a mainstream audience is reduced to those who can meet the exact hardware requirement.

As a result, to investigate new techniques in approximate computation the current solutions available need to be considered and determine what stage of the software pipeline they effect and how they can be more widely applied. This way, we will have the understanding to be able build upon the current research and consider more generally applicable techniques.

This chapter first focuses upon key ideas and problems in the field. I then consider reliability analysis, error-resilience and the common methods used to approach measuring them. Existing AC Approximate Computing techniques which are influential and form the motivation for this research are then covered in detail.

This chapter gives a high-level overview of current approximate computing research, showing which areas I am building on and places my work in that context.

2.1 Key Problems in Approximate Computation

There are three important problem areas in approximate computing which have been identified [115] as discouraging to mainstream acceptance and spread of AC as standard computational technique.

- Finding applicable code
- Lack of generic approaches
- Solution verification

Each is covered in detail below.

2.1.1 Finding Applicable Code

This represents the challenge of identifying particular segments of a target application which could be viable for approximation. This would be code which can be altered without negatively impacting the output of the application. Code-bases are very complex and therefore to find such segments the approaches needed would have to be in some automated tool [136].

To identify these segments of code an analysis of the potential impact to the code-base is required. One proposed way of doing this is through code-injection to determine for different code segments the sensitivity they have in relation to the final output. This is known as the perturbation-output problem [159]. The impact of changes made to the code must be understood to predict how the changes will propagate error through the program, and to determine the affect on the output.

Code segments which are resilient to error are considered ‘robust’ and the extents of that property are widely explored [77, 78, 30, 100, 112]. Works in this area demonstrate the tolerance to error of different patterns of coding [93] and the different algorithms they use. Through this analysis they are able to produce approaches and algorithms for *fault-tolerant* computing [131]. These research approaches all consider execution on systems which are free from external influence, in the real-world their is the concept of AVF (*architectural vulnerability factor*) which refers to the vulnerability of hardware to random changes caused by particle strikes. These random events can also effect the resiliency of a system if they are not handled correctly [117]. Chapter 6 shows a framework for programmer guided resilience analysis to try and reduce the overall workload and allow for targeted optimisation.

This problem of robustness is difficult to approach for different types of applications and at scale. For large solutions, testing every section of code for every possible input can become computationally difficult, with some studies reporting compute times of many days with workloads spread over hundreds of machines [159].

When looking at the limits of resilience you reach a critical error when the error leads to an unacceptable output or crash (see Section 2.3). In the field of security the analysis of errors is very important, one taxonomy on error and dependable computing lays out definitions for many different categories of software error [9]. From that taxonomy we get the definition of *Latent Errors*, ‘Errors that are present but not detected’. An optimal approximation solution aims to reach the best performance by producing only latent errors – a non-trivial task.

As such, the challenge of finding applicable code lies in identifying sections of code which would:

- **Produce a benefit if altered**
  Which can only be identified by measuring the performance of each code-segment.

- **Be resilient to some form of error**
  Which is identified by performing sensitivity analysis of the whole code-block in relation to the final output (the "perturbation-output" problem).

- **Change would not result in a complete breakdown of the application**
  Through criticality testing for the domains of the code-segments being changed we need to ensure they will not result in the erroneous, unexpected or abnormal termination of the program [131].

If this criteria can be met by a code-segment then it is a candidate for approximation.

2.1.2 Lack of Generic Techniques

In areas where approximation is not compulsory, there are a lack of generic approaches for practical applications [139]. Many approaches are highly specified and focus upon limited applications; this results in large overhead, strict hardware requirements (such as the inclusion of specific approximation capable hardware [29]), and custom language extensions [138, 140]. As a result, exploitation of AC for mainstream applications is not possible while the existing modern techniques are incompatible with standard development environments or hardware configurations. This makes many techniques functionally ineffective – especially to deploy at scale without a high development or end-user cost. As a result, exploitation of AC for mainstream applications is not possible while the techniques are incompatible with standard development.

Additionally, the lack of generic approaches means that general tooling to analyse software and resolve the challenge of finding applicable code for specific techniques, are either non-existent or are frequently over specified for specific techniques or a single heuristic [77]. This is one of the causes for developers to implement ad-hoc approximation solutions which are not robustly explored as correct or safe such as screen-space only ray-tracing [108], screen-space ambient occlusion [12], approximate anti-aliasing [105] and approximate mathematical functions [104].

2.1.3 Solution Verification

It is challenging to verify if a program executes correctly across its entire possible input domain – especially in large multivariate systems [132]. A common approach in systems development involves heavy reliance upon testing for verification through hand-selected targets [106], which can result in limited coverage of error testing, and therefore may not identify all problems [67].

Applying approximation increases the scope of unexpected changes in behaviour which may not have been covered by testing. For an approximation technique to be valid, it must fit within acceptable parameters for error and stability, without producing silent data corruption[1]). For a complicated function, it is near-impossible to acceptably ensure verification for every input-output pair, and is made more complex when considering acceptable error bounds. Research and development aimed at containing this problem for certain techniques has been attempted [167], but no consensus for generic approaches has been developed for generic application without human intervention.

In my work I verify solutions within the limited domain which results from the change applied using my framework for guided identification of approximation sites. I also limit work to 32-bit floating-point for most cases and perform isolated code-transforms so that only small areas of code need to be validated. For approximations which are more open-ended (such as improvements to neural network computations in Chapter 7) I rely on verification of the final output rather than the steps in between. They are inconsequential as only the final result has any impact.

2.2 Application

In this section, the standard approaches and considerations for applying AC to a program are presented. Beginning with the motivation for using approximation, followed by a breakdown of when it is possible to be applied. This high-level view of the area will provide the background knowledge which may be assumed in Section 2.4 covering specific AC techniques.

When looking to apply approximation, it is important to consider the motivation for using it. For some intractable problems, an approximation is mandatory [139], in these scenarios a "good-enough" [141] answer is used and for many common problems – these are well documented [33]. For other cases, where it is not essential to use approximation, other factors are incentives to apply the techniques. In this section we will cover the different motivations for using approximation and where they are generally applied.

Approximation Is Required. As mentioned, approximation can sometimes be a requirement. The majority of instances of approximation that are presently encountered are cases of mandatory

[1] See Section 2.3 for more details

approximation. Common use-cases include machine-learning, simulations and low-cost real-number calculations.

For these types of problems AC is a natural solution as the result is not required to be absolutely correct, there is a high-level of tolerance to specific inaccuracy and computing the absolute correct answer would be either impossible, impractical or unnecessary.

The application is error-resilient. Some software has a high-level of resilience to error. This is often the case for applications which rely on sensors or noisy data [148]. These setups have an inherent error-tolerance which can be exploited for an energy or performance benefit.

A common example of this would be a graphics application which renders images to a low-precision buffer. If the difference in absolute value between the actual result and the low-precision result is lower than the viewers ability to perceive it, then the application has reduced the cost of rendering without a negative impact [137].

Error-resilience proves to be a beneficial, motivating characteristic for a programmer to intentionally use AC techniques [30].

Efficiency Gains. Some applications need to increase performance and are willing to pay an accuracy cost for it. This is of particular value to a developer working on limited hardware, such as those developing for mobile devices.

A simple example of this is a developer working on a video encoder and decoder. The user of the mobile device may be willing to accept visual artefacts in the decoded video if it allows the playback of video at an acceptable quality at a lower processing cost.

Quality Control. AC techniques can allow for scalability of the final Quality of Result (QoR). It can be desirable for a program to target multiple performance levels of hardware. Approximation can allow for automatically lowering the performance cost to match the hardware with understood consequences. In this way a developer is able to tune the performance-QoR trade-off appropriately for each target platform.

Once it is established that AC is an option then it must be considered if the problem falls into a classification where it can be reasonably applied.

It is most important that the part of the program targeted for approximation does not result in a *Golden Answer* [161]. A *Golden Answer* is one which must be absolutely correct for any given input and any deviation from this would be a critical failure of the program [159].

If an application does not have a *Golden Answer* then it will have some margins of acceptable error, known as resilience[2] [30].

Once resilience has been established the approximable properties which an AC technique could exploit can be considered. These include:

[2] Resilience is further explored in Section 2.3 of this chapter.

- **Resilient Results** When a program contains stages where a bounded level of error will not negatively effect the QoR. This is a common property of programs which use noisy or unreliable data-sources. Such as sensors in mobile devices which may disregard anomalous data outside of a fixed range or programs which only require a low-precision output to function correctly [148].

- **Stochastic/Probabilistic Results** This property is exhibited by a program which will converge onto the correct result, in possibly an infinite number of steps. A common example would be Monte-Carlo based probabilistic algorithms [57].

- **Self-Correcting Stages** Similar to stochastic programs, this property is expressed through error from approximation which is then later "corrected" by a more costly update. This pattern is common to algorithms that rely on prediction from uncertain data [148].

- **Fail-Rare Or Fail-Small** If a program can support small errors or infrequent errors, it will fall into this category [30]. Real-time software often falls into this category. Where a small error will only have a small impact on the final result and is acceptable to the end-user. A common example would be a video decoder which may have incomplete data resulting in small error per-frame output.

With this it is possible to reasonably establish which type, if any, AC technique is suitable for a program.

Approximation techniques are heavily reliant on understanding error propagation and resilience. As this behaviour is expressed differently at different levels of code abstraction, AC techniques are grouped into different levels; operation-, type- and algorithm-level [30].

- **Operation-level.** Low-level hardware or type interaction overrides, such as voltage scaled hardware [76].

- **Type-level.** Approximate data representations, such as bit truncation or lower-precision types [38].

- **Algorithm-level.** Skipping tasks, such as loop perforation or memoisation as well probabilistic approaches such as Monte-Carlo simulation [109].

Knowing which level of approximation is best to approach for a problem, informs how to measure error, as well as which techniques would be suitable and how viable they would be.

Understanding of the key motivation, use-cases and classes of techniques is important to the selection of approximation techniques. In my work I show examples of work at all three levels of approximation. In Chapter 6 I show operation level tweaking of existing functions to selectively remove code that is providing more accuracy than required in resilient use-cases. In Chapter 5 I show type- and algorithm-level approximations when different internal precision levels are used, as well as different curve-fitting algorithms to find implementations of mathematical functions with different amounts of error for use in a fail-small algorithm[3]. And finally, in the last work chapter, Chapter 7 primarily algorithm-level approximations of known activation functions are used to improve the performance of self-correcting machine learning algorithms.

2.3 Resiliency

A recurring problem in AC is defining an objective quality metric from which deviance or error can be measured [139]. As a result, determining whether a section of code is safe to approximate is difficult. Analysis techniques which attempt to measure the error and determine limits on the amount of error that can be introduced to a section of code are referred to as *Resilience Analysis*.

In this context, the resilience for a part of an application is it's maximum tolerance for error before the application no longer performs adequately. Some promising modern approaches have looked at measuring this through bit-level fault-injection [77, 78]. This allows for independent analysis of the tolerances of each section of code, and gaining a reasonable understanding of the magnitude of error in a system when it is produced at different code-positions. Through experimentation (within known input ranges) this can be used to produce an estimate of acceptable error bounds.

When approaches are looking to measure resilience, applications are grouped into two categories, those which are inherently resilient and those which are not [30]. Inherently resilient applications are often those which implement software containing compulsory approximation, resulting in some leniency in the final results. Non-inherently resilient applications are those where we are interested in *if* that leniency exists. Resiliency is easier to measure when it is inherent as any working application can be estimated from the programmers 'expectations' [48]. 'Expectations' is an approach which allows for the relaxation of accuracy constraints by taking the assumptions of implied error in the types and mandatory approximation techniques being used and using them to assume a specific minimal level of resilience within specific code section. In non-inherent applications the maximum errors which are acceptable must be determined in more complex ways, as the code is written to be accurate and no assumptions can be made.

A useful tool for determining the objective quality for non-inherently resilient applications is the concept of a *Comfort Zone* which was proposed by M. Rinard [133]. He proposed that analysis should be performed within fixed acceptable input ranges for a function or part of a code-base. Within this range there should be robust testing and acceptable bounds can be established. This was originally proposed to allow for the separation of core functionality of an application from unstable elements, but the concept holds for determining acceptable bounds

[3] In this case, the fail-small scenario is converting from 64-bit to 32-bit, where small errors are likely to be cancelled out by being unable to be represented at the lower precision.

of input for other transformations and approximations and is further explored in other papers, sometimes related to *Good-enough Computing* [132].

When testing the *Comfort Zone* input range to determine the resilience, it is to determine if the injected error results in error in the output of the application that is either silent, detectable or critical [159]. A data change which is silent means that error has not had a measurable impact on the output meaning the *Golden Answer* is maintained, a detectable error is one which does affect the output to some degree but within acceptable limits and a critical error is unacceptable or causes other problems which prevent the output being measurable (e.g. segmentation faults).

If testing is thorough, then it is possible to calculate the degree by which different parts of the application can be changed if there is an appropriate way to understand the results of the application – with caveats on the type of approximation technique that will be applied and the type of data that is being accessed but further work for a standardised approach is still needed.

2.4 Current Approximation Strategies

The problem of approximation has been explored by researchers from diverse backgrounds and different areas of computer science, this is reflected in the variety and difference of the many approaches.

A major difference between techniques is the strictness they take to performing the transformations. There is also a wide discrepancy in which level of abstraction each approach chooses to apply its changes to produce the approximation. The techniques vary, with the lowest level requiring physical changes to the hardware, and other techniques for every stage of compilation, and even some designed to exploit behaviour in the hardware drivers.

These differences in approach will be expanded on below for each technique that is described. With the analysis and characterisation of select techniques in the field I hope to show clearly where gaps exist in the current techniques.

2.4.1 Hardware Approximation

Many of the optimisation techniques which use approximation have attempted to explore changes in hardware to facilitate the use of approximation in the source-code. This tends to have good results [140] but this limits to the possible distribution of the techniques beyond machines with specific and uncommon hardware – unless the technique is adopted into hardware by a major distributor.

As the aim of our proposed research is to find solutions to key problems which prevent the safe and reliable application of approximation in *general use*, any hardware changes are out-of-scope, and potentially adverse to the goals.

For thoroughness, the reader can be directed to the body of work on this subject, which appears to be more active than current software developments in approximation [76, 168, 110, 140, 148, 27, 92]. For now, the reader only needs to understand that these techniques can be effective and they provide an additional avenue to approach the problems at the cost of being very limited in applicability in the main-stream.

2.4.2 Mandatory Approximation

Approximation, in the general sense, has been very common in computing for a long time in the form of floating-point numbers. Floating-point representation is considered one of the oldest forms of approximation that is ubiquitous in modern computing [165]. Without approximation it would be less practical to represent real numbers in real-time applications. Using floating-point gives programmers the option to choose an arbitrary level of accuracy that is suitable to solve a problem with a very strictly defined behaviour and representations [3].

When considering mandatory approximate computing, it is for the cases where hard real-time constraints clash with computationally difficult problems without a "golden answer". This has resulted in some forms of approximation being reasonably well-defined and integrated into specific paradigms [69, 109]. Floating-point numbers have been mentioned, but other examples of mandatory approximation include: real-time graphics [96] (which are often not formally defined), machine learning [128, 147], probabilistic algorithms [57, 100, 63, 46] and sensor-data handling [148, 61].

Approximation has often been seen as a negative approach for introducing error into a system when it is not absolutely required, but it is surprisingly spread throughout modern systems. This is important to recognise to build a case for bringing AC into mainstream software development. As it is now, approximation is mandatory in some areas for reasonable performance but it is strongly bound to those areas and application of it outside of those specific regions can be dangerous due to the lack of verification processes [139].

Real-time graphics has shown that there is an abundance of use-cases that do not have to be formally defined to be safe and effective, and that in the real-world ad-hoc usage approximations may be made safe through simpler verification to allow for a larger optimisation space than stricter definitions in other mandatory approximation techniques.

2.4.3 Mathematical Approximation

Mathematical approaches are one of the more established areas of approximation, with floating-point representation being a good example. When IEEE floating-point hardware was becoming standard on CPUs, J. Blinn introduced approaches which allowed for the approximation of a few simple mathematical functions [18] which are still valuable today. Based upon exploiting how data is stored in IEEE floating-point representation he was able to produce small functions which outperform the standard implementations with a varied loss of accuracy.

More modern approaches, such as MetaLibm [97] have approached similar problems. MetaLibm seeks to map the entirety of the C Math Library [88] into approximate functions with a degree of control over the precision. It represents some of the most robust function replacement available in the field, although the approach is limited to functions which can be represented as being continuous due to the reliance on curve-fitting and validated Taylor models.

Even with the limited scope of MetalibM, some of the results are very accurate to the functions that are being replaced, with performance improvements of upto 5.8x at arbitrary levels of precision.

Strict mathematical approaches provide solid evidence for the replacements that they create. This makes testing and evaluation much simpler. Mathematical transforms are also simple to represent at different levels of abstraction, allowing for a more generic approach, particular when generating full replacement functions. This is of particular value to our aims as it validates function replacement and shows the viability in performance and applicability of function/algorithm-level approximation approaches. This is explored in more detail, with robust examples in Section 3.2.

2.4.4 Precision Scaling

Changing the precision of the data-types used to represent data in an application is an obvious early target for approximation, especially when using floating-point.

One approximation system developed to target this area of techniques is EnerJ [140], with the aim to reduce power usage through approximation. This system identified the need to keep approximate data separate from accurate data. By keeping the data expressly separate they can store the approximate data in lower-power approximate-precision storage hardware.

Allowing a separation in the storage hardware of the data forces the programmer to explicitly control the transform from approximation back into regular data flow. Control of the approximate hardware is exposed to the programmer through an extension for the Java [71] language type-system and introduces specific annotations to the language to allow for control of the behaviour of the approximations.

The software aspects of EnerJ are implemented as a language extension for Java, working on type-level approximation. It relies on strict verification through static source-code analysis. Giving express control of approximate hardware and integrating the tools into the Java type-system EnerJ is shown to be effective at allowing basic precision scaling for approximation.

This approach is somewhat limited for large scale deployment because of approximate/low-power storage hardware is not widely available as well as the technical cost involved in upgrading the language that is being used and annotating an existing code-base.

Additionally, with EnerJ only targeting type-level approximations it still requires large amounts of human effort to implement the approximate algorithms which use the approximate types shown in the paper. The introduction of an automated system to generate the replacement functions to handle the algorithm-level changes would reduce the cost of implementing the software side of this technique, if sufficient guarantees could be assured in the static analysis of the generated result.

In my work I have taken the influence of precision scaling and applied it to manually select the precision of approximations at different stages through their algorithm. This can be seen in Section 5.6 where I use a similar approach to produce correctly rounded 32-bit mathematical functions implementations by scaling the internal precision of the polynomials being used at the root of their algorithm.

2.4.5 Memoisation

Memoisation is a technique that allows the skipping a call to a function with a specific input and returning a stored result if that function has already been called before with the same input. The function being memoised, usage pattern, and the hardware being used for execution all change the scale of performance benefits this may produce and decide whether or not this is an efficient optimisation for the target.

Applying this simple concept to AC gives two different areas to target. It is possible apply approximation to the output, which can be done by interpolating between results and varying the granularity of the results tables that are being used. This alters the QoR of the functions which are being called. Or, it is possible to allow for approximation on the input of the function, so that similar inputs produce the same output.

A key paper in this field advocates for a *Clumsy Value Cache* [92]. This technique is specifically targeted for mobile GPUs. They note that the high-precision outputs for different sections of a fragment shader have a high level of variance. As a result they look to only memoise small sections of the fragment shader which fit into a very specific criteria to maximise the chance of the multiple hits to the same stored table index. They also reduce the number of matching bits for an input to be considered the same as another input. With this strategy any similar input values for sections of a fragment shader can be replaced at run-time with the stored results. This is particularly effective at reducing the high-cost of texture reads but can also degrade the QoR more because of high-frequency data that is often stored between nearby UV-coordinates in a texture.

The results of generating an efficient clumsy value cache shows a reduction of around 13.5% in the number of instructions that must be executed on the GPU. The drawback of this specific approach to memoisation, and lots of the techniques discussed in this chapter, is that it requires a specific change to the way that code is compiled for the GPU and potentially specific hardware to be effective at changing the run-time performance cost due to the high-cost of frequent reads and writes to the value cache. A compilation change would require it to be compatible with existing driver software if it is an aim of the developer to integrate these changes into existing code which may be reliant on specific behaviours of the driver that is currently being targeted.

In my work I am trying to avoid specific hardware or compiler changes as this reduces the ability of the technique to be generically applied to existing software and places constraints on where the resulting software can be used. However, the approach to use a clumsy value cache is interesting to our research aim – if applied in a more general way.

This area of approximation influenced our development of table-based mathematical function implementations, although in the end I have not fully applied approximate table look-ups.

2.4.6 Function Generation

Full function generation has taken a few forms, commonly this is performed through mathematical pattern matching and replacement (as is found in optimising compilers [103]). Others have taken the form of sub-function replacement, like those of H. Massalin's *SuperOptimiser* [107], the basis for modern peephole optimisation with a search space of possible replacements from which a new sub-function can be constructed with preferable performance characteristics. Building upon these ideas there have been approaches in randomisation to find function replacements which take into account accuracy when considering the replacement [170].

With the rising popularity of machine learning, interest in applying approximation into function generating techniques are also on the rise [160, 169, 116]. While the results of this new work do produce programs that are effectively building functions, the lack of control over the production and inability to modify them in specific ways after they have finished the learning process makes them inappropriate for standard software development.

Of the approaches in this category, the approach for the *SuperOptimiser* is most relevant to the approaches that I would consider for a generic approach. However, that method is limited by the number of instructions it is possible to process in reasonable time for each *peephole* that it is trying to rewrite. Which must be solved to build upon this for a generic approximation solution.

2.5 Discussion

From covering the current material in the field, I have been able to identify approaches which are more or less applicable to the real-world and may help form the basis of a new technique which will be more suitable for mainstream adoption.

I found that to be suitable for mainstream adoption, I should be targeting a solution which works at either the source-code or compilation stage with minimal changes to the existing language. This will allow the users to continue to work without having to do significant changes to existing code-bases.

The new technique should work at the function-level. This aids in the analysis of the code, making finding applicable code sections simpler and gives the potential for larger-scale optimisations which lower-level approaches such as peephole optimisation, and approximations in that area, would miss due to the limited scope.

There is also a need for generic tools for measuring error in existing solutions so that areas of code that can be approximated are easy to identify and validate.

This research guides the work in the rest of the thesis. I have developed a framework (Chapter 6) for detecting resiliency and applicability of approximation at a function-level in existing code-bases, thus averting the difficult problem of having to generate entirely new functions from scratch. I provide a background of the existing error in the functions we want to approximate (Chapter 4) to understand the landscape of the problem being targeted for approximation (and justify that any approximation in this area is acceptable). Finally these approaches are applied to improve the implementations of a number of functions used in mathematical computing (Chapters 5, 6 and 7).

This places our work in line with the current trends and findings in approximate computing so that any further work with the tools or results presented here can be easily adopted by those working in the field.

Chapter 3

Background

Approximation lies at the heart of any numerical computation which makes heavy use of floating-point numbers.

Here the development of floating-point and how approximations have been core to the engineering of mathematical libraries is covered in detail, as well as covering any relevant features and topics related to numerical computing for mathematical functions. The aim is to give the background information needed to provide the context for our work in later chapters.

I adopt standard definitions of common mathematical terms and summarise these for reference in Appendix A.

3.1 IEEE-754 and Numerical Computing for Elementary Functions

The IEEE-754 Standard for Binary Floating-Point Arithmetic is the de-facto standard for the representation of *real* numbers in computer systems. Among other things it defines a binary representation for real numbers, the behaviour of arithmetic operators, and the behaviour of additional mathematical functions. These functions are the building blocks for a huge range of numeric programs and so provide a natural focus for optimisation and approximation.

The standard's stated aims include:

> "Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity"

and

> "Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs" [2]

In its goals, it has been highly successful: modern software is ported between devices and used with floating-point libraries with impunity and little noticeable consequences despite many software developers have no formal training in numerical methods.

Later revisions of the standard included guidelines for the development of suggested elementary functions due to their importance for numerical computing with floating-point numbers.

3.1.1 The Evolution of IEEE-754

The standard began with an original proposal [89] and subsequently has evolved through two further revisions. The hardware implementations and software libraries that support it have evolved in tandem and early work seems to have hugely influenced the modern implementations which I have studied. Here the major milestones are summarised as context for a later discussions on the performance of modern floating-point libraries.

The 1960s to late 1970s were the early years of generic automated mathematical computing. The only widely available generic numerical computing solutions were very specialised and limited software-based [55] solutions and access to specialised hardware was very limited. It was not until near the end of this period that hardware implementations began to be discussed and seriously implemented.

The software that was available at the time was developed by scientific institutes [47, 58, 16] and universities [58, 51, 52] all of which followed their own guidelines for accuracy, performance, and reliability, making their own decisions about what functionality to include. As a result, much of the software was non-portable [58] and made specific to the tasks that each institution were facing [17].

At the end of the 1970s, the development of specialised hardware for some key mathematical functions began [149, 134]. For there to be generic hardware to process mathematical functions it was also required that the standard for representing the real numbers they would generate was agreed upon. This was part of the reason for the formation of the IEEE-754 working group and the subsequent standard [144]. By the early 1980s hardware implementation of some of the trigonometric functions began to be widespread. One example is the 8087 instruction set [124] which contained CORDIC [124] based implementations of tangent and arctangent.

These hardware implementations were significantly less accurate than the standard (which followed them) would later request and while these implementations were in development at the same time as the IEEE-754 standard, they do not conform to the modern interpretation of the standard. They are significantly less accurate than correctly rounded [121] solutions. In the early 1990s, some CORDIC implementations were replaced with polynomial implementations to improve accuracy [50].

The *IEEE-754 standard for Binary Floating-point arithmetic* was officially released in 1985, although major hardware manufacturers had been preparing their own conforming hardware starting years earlier. There was some disagreement during the development of IEEE-754 as

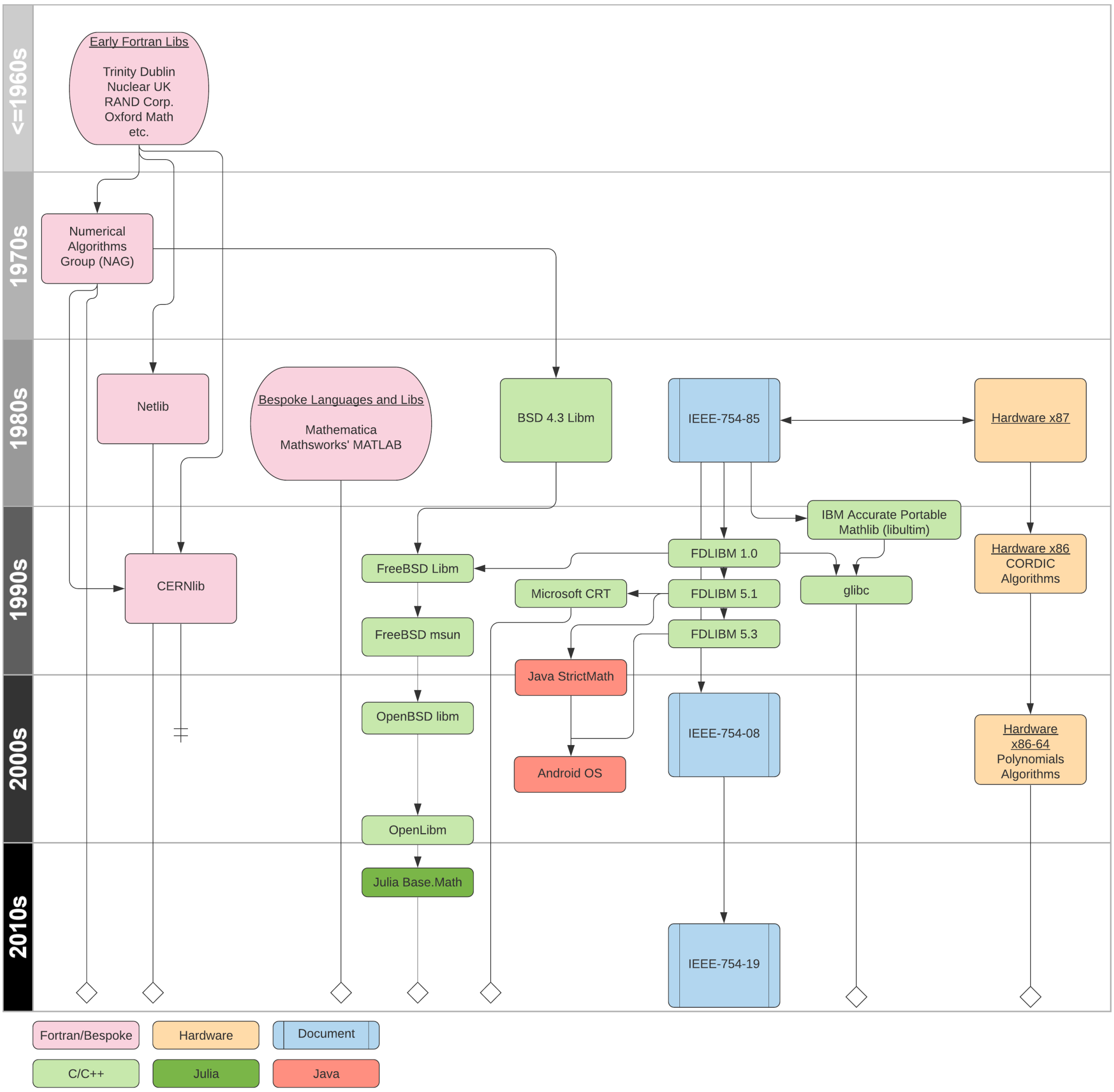


Figure 3.1: A brief outline of the relationship and development of modern mathematical libraries.

Rounding Mode	Description
Round to Nearest, Ties to Even	The default and most widely used rounding mode
Round to Nearest, Ties Away from Zero	Similar to the default, but ties round to $\pm\infty$
Round Towards Zero	Rounds towards zero. Effectively a truncation.
Round Up	Rounds towards $+\infty$
Round Down	Rounds towards $-\infty$

Table 3.1: The provided rounding modes of IEEE-754.

many manufacturers had concerns about the complexity and performance costs of the implementation [171], but in the end, the standard generally required operations to have the best possible accuracy for the chosen representation. To assist in the adoption of the standard, in the early '90s, Kahan and his students at the University of California, Berkeley worked together to produce FDLIBM as a reference numerical computing library using IEEE-754 floating-point numbers. FDLIBM has proven hugely influential to the development of floating-point libraries and later I show how its legacy can still be seen in modern source code.

The 2008 revision of the standard (originally planned for 2001) included the significant addition of "Recommended correctly rounded functions" which are a focus of this work and covered in detail in Section 4.

The standard was again updated in 2019 with some minor changes and improvements to clarity in some of the language. Notably, the term 'additional mathematical operations' was introduced to replace 'recommended function'. The 2019 revision is the most recent version of the standard at the time of writing.

3.1.2 Rounding Modes

In the IEEE-754 standard, there are five rounding-modes to choose from, they are shown in Table 3.1. The standard rounding mode is 'Round to Nearest, Ties to Even' which is intuitive with how rounding numbers is usually handled in the real world. Other rounding modes are useful for specific mathematical techniques and to assist in interval arithmetic [68].

In this dissertation 'Round to Nearest, Ties to Even' (RtoN-TtoE) is used, as it is the default and most commonly used approach in most systems. Any other rounding mode usage will be explicitly signalled.

When it is stated that an answer is correctly rounded, it means that in the context of the rounding mode. It means that the resulting value has been rounded to the nearest representable value to the true value as dictated by the rounding-mode.

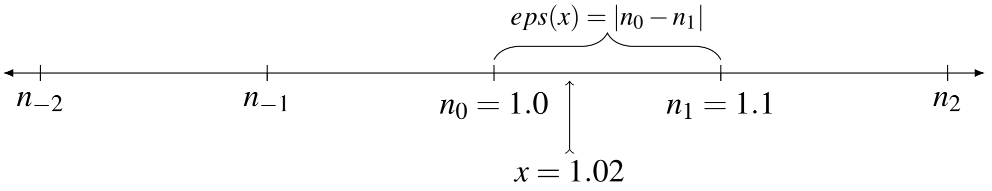
3.1.3 Units-in-Last-Place and Correct-Rounding

The term *correct-rounding* refers to selecting the best representable floating-point value for a given real number, in a specific rounding mode. Unit-in-last-place (ULP), are used to measure error in terms of the smallest possible change of value for a number of that value. The ULP error for the Round-to-Nearest rounding mode , where n_0 and n_1 represent the two straddling floating-point representable values that could be rounded to, can be given as:

$$\text{Round-to-Nearest ULP error} = |(x_{rounded} - x_{real})|/|n_0 - n_1|$$

When comparing a real value to the possible representable values in floating-point a result of ≤ 0.5 ULP error is produced when rounding the real value results in the correctly rounded representation. As such, stating the resulting value has ≤ 0.5 ULP error, is often the same as saying the value is correctly rounded.

The diagram below shows a real value x which is of a higher-precision than the floating-point numbers represented on the number line. If it is desired to convert x to a number on the number line we must select a value that we can represent. In this case that would be accomplished by rounding to one of the two straddling values of x: n_0 or n_1.



The correctly rounded value can be selected by using the ULP error equation above:

If we choose n_0 as the rounded result:

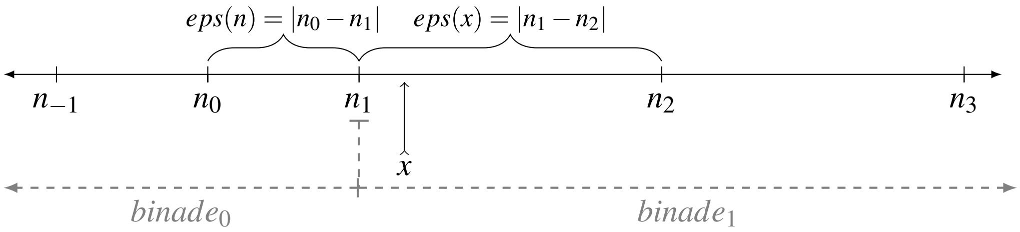
$$\begin{aligned}
\varepsilon_{n_0} &= |(x - n_0)|/|n_0 - n_1| \\
\varepsilon_{n_0} &= |(1.02 - 1.0)|/|1.0 - 1.1| \\
\varepsilon_{n_0} &= 0.02/0.1 \\
\varepsilon_{n_0} &= 0.2
\end{aligned}$$

Or, if we choose n_1:

$$\begin{aligned}
\varepsilon_{n_1} &= |(x - n_1)|/|n_0 - n_1| \\
\varepsilon_{n_1} &= |(1.02 - 1.1)|/|1.0 - 1.1| \\
\varepsilon_{n_1} &= 0.08/0.1 \\
\varepsilon_{n_1} &= 0.8
\end{aligned}$$

This shows that n_0 is the correctly rounded value for x because $\varepsilon_{n_0} \leq 0.5$.

There are some pitfalls to this method of ULP error measurements. In particular, care must be taken when rounding at the boundaries of IEEE-754 floating-point binades[1]. Consider the example below where n_1 is the correctly rounded result and x is a *real* input:



The value of n_1 sits at the border of a binade, the values to the left have half the distance between them to those on the right. If we use the ULP step size of x and repeat the calculations above we would find that n_1 is still the closest result, but both neighbouring values would be ≤ 0.5 ULP error - which cannot be true. To avoid this error, the *eps* function used must return the ULP error in relation to the final floating point number. If this is performed correctly in the above example x will have a ULP error of > 1 for n_0.

Due care must be taken when calculating ULP error to ensure the error is in the correct space and binades do not accidentally cause a result to appear to be correctly rounded when it is not.

3.1.4 The Additional Mathematical Operations

All revisions of the standard since 2008 specify a set of required functions. In the current (2019) standard these are referred to as “Additional mathematical operations” [3] and previously in the 2008 revision “Recommended correctly rounded functions” [2]. This set includes the exceptions, cases, domains and special values for these functions as well as stating the requirement for correct rounding.

In this work, I focus on a subset of these functions: the single-input exponential, logarithm and trigonometric functions. These functions were selected because they are: a) commonly used in a wide range of applications including graphics and neural networks (See Chapter. 7); b) transcendental (cannot be expressed as a finite number of algebraic operations) and thus challenging to implement correctly; c) possible to fully empirically evaluate in 32-bit floating-point precision.

In the (2008) revision the terminology of recommended function could be considered confusing with an alternative interpretation that the correct-rounding is only recommended. However, there is evidence of the modern interpretation in the text:

> *“inexact: Functions should signal the inexact exception if the result is inexact. Functions should not signal the inexact exception if the result is exact.”*

and

[1] A binade is the set of all IEEE-754 numbers which have the same exponent.

> *"A conforming function shall return results correctly rounded for the applicable rounding direction for all operands in its domain. The preferred quantum is language-defined."* [1]

This means that a language/library must signal inexact answers and may choose a larger ULP error to use for correct-rounding if required. None of the implementations investigated later take either of these routes.

This is further refined to be clearer that correct-rounding is the goal in the 2019 revision of the standard where it states that the standard requires that these additional mathematical operations are correctly rounded:

> *"A conforming operation shall return results correctly rounded for the applicable rounding direction for all operands in its domain."* [3]

3.1.5 The Ongoing Influence of FDLIBM

FDLIBM was written by William Kahan and his students at Berkeley in 1994 as an early reference implementation of elementary functions using IEEE-754 floating-point[2]. The library describes itself:

> *"FDLIBM is intended to provide a reasonably portable (see assumptions below), reference quality (below one ulp for major functions like sin,cos,exp,log) math library."*

[151]
Although not mentioned above, the library is only accurate to one ULP when using the round-to-even rounding mode. This conformed to the IEEE-754 standard at the time since the requirements for correct rounding were not added until the 2008 revision.

Despite the library being intended as a reference, it saw a lot of use, especially as the mathematical library provided with some Linux distributions, some Java distributions and Java's StrictMath package [90][3]. Some of this usage continues to this day with project dependencies on particular versions of FDLIBM requiring even modern projects like Android to continue to ship this package [70].

A tale of FDLIBM in 2020

[2] There is an earlier implementation of an IEEE-754 compliant maths library called IBM Accurate Portable Mathlib (also known as Ziv's libultim) which provides correctly rounded results. It went on to be part of the early basis for glibc but did not see the wide-scale adoption of FDLIBM [37]

[3] Java's specific non-compliance with IEEE-754 is noted with some passion by Kahan in this presentation: `http://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf`

In the writing of this work developers at Google working on the Android Operating System were contacted to correct a mismatch in headers for version 5.1 of FDLIBM which were being used with version 5.3. This is a very minor mistake, but one that appears to have been in the Java Math source code for over twenty years. Another interesting quirk of the legacy of FDLIBM being adopted into libraries is that the 'libc' used in Android included a 'math.h' header file referencing FDLIBM 5.1 despite all related FDLIBM code in that library having been replaced. This issue has also now been resolved[4] [70].

.

This continued use of FDLIBM is far from ideal as the final release (version 5.3) was made around February 2004[5], four years before the release of the (2008) revision of IEEE-754 which requires correct-rounding.

This means that when this library is being used in programs purporting to be IEEE-754 compliant—it would only be compliant with the latest standards by coincidence. This is particularly true for the jump from IEEE-754-1989 to IEEE-754-2008 where the standard was significantly expanded upon.

Other libraries have continued to use implementations of functions directly taken from FDLIBM. This is evident from the source code of FreeBSD msun [56], OpenLibm [86], Julia [14] and glibc [65]. Section 4.3 also shows how the error distributions of some closed-source libraries suggest the same provenance.

The influence of FDLIBM on the development of functions in mathematical libraries means that one ULP error-bounds seems to have become the standard for acceptable error.

Not only does this not comply with the standard but introduces the problem of correctly rounded libraries having to compete with a widespread "good-enough" solution—to quote a Julia core developer in the Julia Slack channel on the topic of switching to correctly rounded elementary functions: "Do you want to answer the thousands of e-mails we will receive about the performance loss?".

There are two implications of the fact that one ULP error-bounds is accepted (or unnoticed) by most applications. Firstly, it provides evidence that approximation is acceptable in many cases, which allows for an optimisation space between zero and one ULP error (which is taken advantage of in Section 6). Secondly, it suggests that there are latent errors in existing systems that might someday cause an issue. This is explored in Section 4 showing the impact on neural networks arising from a lack of portability.

[4] The commit for this change with the updated comment and removed redundant 'define' can be found here: `https://android-review.googlesource.com/c/platform/bionic/+/1430135/1/libc/include/math.h`

[5] An exact date is difficult here as the dates in the header are inconsistent between different sites hosting the library.

Correctly rounded libraries do exist, such as CRLibm (Correctly Rounded Libm) [35]. But this correct-rounding comes at a performance cost and CRLibm has yet to see wide-scale adoption. Figure 3.1 shows the relationship and development of the common mathematical libraries in use today. This gives an idea of the influence of the different libraries, such as FDLIBM, over time and how they have influenced each other to reach the current state of play.

3.2 Approximating Real Functions

Providing a correctly rounded implementation of a real-valued function is in itself an approximation process. Correct-rounding is simply the best possible approximation given the limitations of the underlying representation. There is therefore lots of overlap between the methods used by those implementing floating-point libraries and the methods used for more extreme approximations.

Here existing techniques are surveyed for building approximations which are then later exploited in this work.

3.2.1 Range Reduction

A periodic function has many values in its range that map to the same value in its domain and so considerable benefit arises from first reducing the range to a single cycle of the function.

Consider the function $f(x)$ which has a valid input domain of $[0,\infty]$ but its output is a repetition of the $[0,2]$ range. To get the best precision possible for the output we would want to calculate $f(101.5)$ as $f(1.5)$ as this would let us use smaller floating-point numbers which are more accurate.

This reduction from 101.5 to 1.5 can be naively implemented as

$$\text{reduce}(x) = x - (\lfloor \frac{x}{2.0} \rfloor * 2.0)$$

This is an implementation of ‘additive reduction’, the general form of this reduction is $x_r = x - kC$ [118].

Naively, this seems to be a good solution. Unfortunately, working with computers always brings problems of limited precision—especially when working with floating-point. As the input value of x moves away from zero there is a greater chance that the rounding after each operation results in an incorrectly reduced argument due to the floating-point step size increasing and the reduced chance of the result of any operation being exactly representable in the given floating-point precision.

For a range-reduction algorithm that takes only small input values and has small values for k and C this is less likely to cause problems because the error will be smaller. It can also be partially remedied in some configurations by increasing the internal precision of the calculations to be large enough that the rounding error is insignificant after the final result is converted to the original lower-precision, for example:

If we wanted to range reduce into $\pi/4$ quarters, as is common in many trigonometric function implementations, a naive 32-bit implementation in the zero to 2π range results in $1.8879 * 10^7$ incorrect range reductions. If we perform the same operations in 64-bit and then round to 32-bit precision, it results in zero incorrect results.

However, for the full binary32 floating-point input range there are $1.070 * 10^9$ incorrect results for the 32-bit implementation and $8.394 * 10^8$ for the 64-bit implementation.

One improvement available is the FMA (Fused Multiply-Add) instruction.

FMA instructions can perform a multiplication and an addition in one step. Combining these operations causes the floating-point rounding which occurs after any mathematical operation to only happen once instead of twice. This further reduces the chance of incorrect rounding.

Using FMA can change output of a mathematical operation and as such is documented in and C/C++ to be controlled through the `FP_CONTRACT` pragma so that the engineer can specify which types of operations are desired.

In a short investigation I found that many modern compilers do not always respect this specific flag as expected. Microsoft Visual C++, gcc and Clang were tested and for my test cases I was only able to cause FMA instructions to be output from simple arithmetic when flags (such as `-mfma` and `-ffp-contract=on`) were set at the compiler level.

There is also the `FP_FAST_FMA` constant which is a requirement in C++11.[6] This flag can be defined to inform the engineer that FMA instructions execute faster than the multiply and add operation it would be replacing, for the target architecture. This is important to know if you care more about speed than accuracy. However, this flag appears to have fallen out of support for some compilers. Tests showed it appears to not be defined in the latest release of Clang (11.0.1) whereas it was defined in the latest release of gcc (10.2) for the same code and architecture target.

I present this information to the user to show the current state of accurate mathematical computing is one where the tools relied on are not always behaving as we would expect, and care must be taken when implementing the algorithms shown here to ensure that your compiler is producing the instructions you expect. In this dissertation where FMA are mentioned I have verified the output instructions and ensured that the hardware used has this capability[7].

Rounding without FMA:

$$x_r = \text{round}(x - \text{round}(kC))$$

Rounding with FMA:

$$x_r = \text{round}(x - kC)$$

[6] It is listed within the <cmath> header synopsis.

[7] It is fairly uncommon for a modern CPU to not have support for FMA instructions in 32- and 64-bit, support for larger data types is more varied.

This upgrade of internal precision is a great benefit for many inputs, but does not solve the problem completely when dealing with the full range of floating-point inputs. To reach the accuracy required to handle the full input domain, more complex algorithms are required.

Cody-Waite Reduction

The Cody-Waite Reduction [32] is a method to improve the working precision of the reduction by splitting the *kC* calculation into two parts using two constants, C_1 and C_2 instead of one. So that the new calculation is:

$$\text{where} \qquad C = C_1 + C_2$$
$$x_r = x - kC_1 - kC_2$$

kC_1 is a number exactly representable in the floating-point format and therefore will not result in any rounding error. This changes the rounding error formula from

$$x_r = \text{round}(x - \text{round}(kC))$$

into

$$x_r = \text{round}(x - \text{round}(kC_1) - \text{round}(kC_2))$$

Which due to kC_1 being exactly representable becomes:

$$x_r = \text{round}(x - kC_1 - \text{round}(kC_2))$$

The rationale behind this approach is that the input, x, is exact but the kC component of the basic range reduction cannot be represented exactly in finite precision and therefore will induce some rounding error, whether it is significant to the final result or not. As the input value x increases, more data will be lost in the subtraction due to the inexact kC component. The Cody-Waite reduction method aims to reduce this error by simulating higher-precision arithmetic. The input does not benefit from higher-precision as it is already correctly represented, only the constant subtraction.

With this approach, C_1 is generally much larger than C_2, and rounding only occurs on kC_2, so any rounding error which may occur is significantly diminished. This approach gives greater than working precision and can be generalised to n-number of terms to try and increase the precision even further as needed.

Even more accurate reduction

This can be even further improved on, but with algorithms which are not so simple to summarise as those presented here. More complicated range-reductions are used when even a number of steps of Cody-Waite reduction is not enough to reach the working precision needed, or for when dealing with large ranges of inputs for scaling factors which do not nicely fit into the Cody-Waite scheme.

For many algorithms in the standard mathematical libraries analysed in this dissertation they use an approach called Modular Range-Reduction which is fully described in fixed- and floating-point in a paper by Muller et al. [36].

3.2.2 Piecewise Functions

While some approaches to approximating a function are cheap, they cannot always be made accurate enough to replace the whole domain of a function. As such, many implementations make use of multiple sub-functions to reconstruct the target function.

Careful choice of sub-functions allows authors to hide the discontinuity between the pieces to meet requirements of differentiability at any point, continuity, or monotonicity.

This can be seen in the Julia 32-bit `sin` function. Figure 3.2 shows the location of one of the discontinuities of this piece-wise implementation. Without looking closely the discontinuity is not visible, however, the closer views in that figure show each individual 32-bit floating-point input at the discontinuity and compares it to the 64-bit implementation in the same input range. Although there is a minor gradient change at the discontinuity the correctly rounded output is still -chosen. The same figure also show an error graph of the 32-bit implementation at the discontinuity. The graph show that error begins to increase as the discontinuity approaches but never significantly enough to change the output detrimentally.

3.2.3 Table-Based Approximation

Some approximations, where accuracy is not the first priority and memory access is fast, may opt for a lookup from a table of pre-computed answers. Tables can be of an appropriate size to meet the accuracy and memory usage requirements of the application, and extra information can be added to assist in reconstruction.

These methods vary from a direct lookup into a finite table, to more complex table which model the interpolation between data-points to best match the function which is trying to be approximated.

Techniques like this are used when memory is available and access is fast-enough to produce equally good results as direct computation in a similar amount of time. These techniques are also used when the mathematical function required is computationally expensive to calculate and the result of a single lookup is used multiple times in different places. The latter case is commonly seen in GPU-based algorithms where functions of two inputs are stored in 2D textures and the specialised image processing hardware can be exploited to implement the interpolation between results.

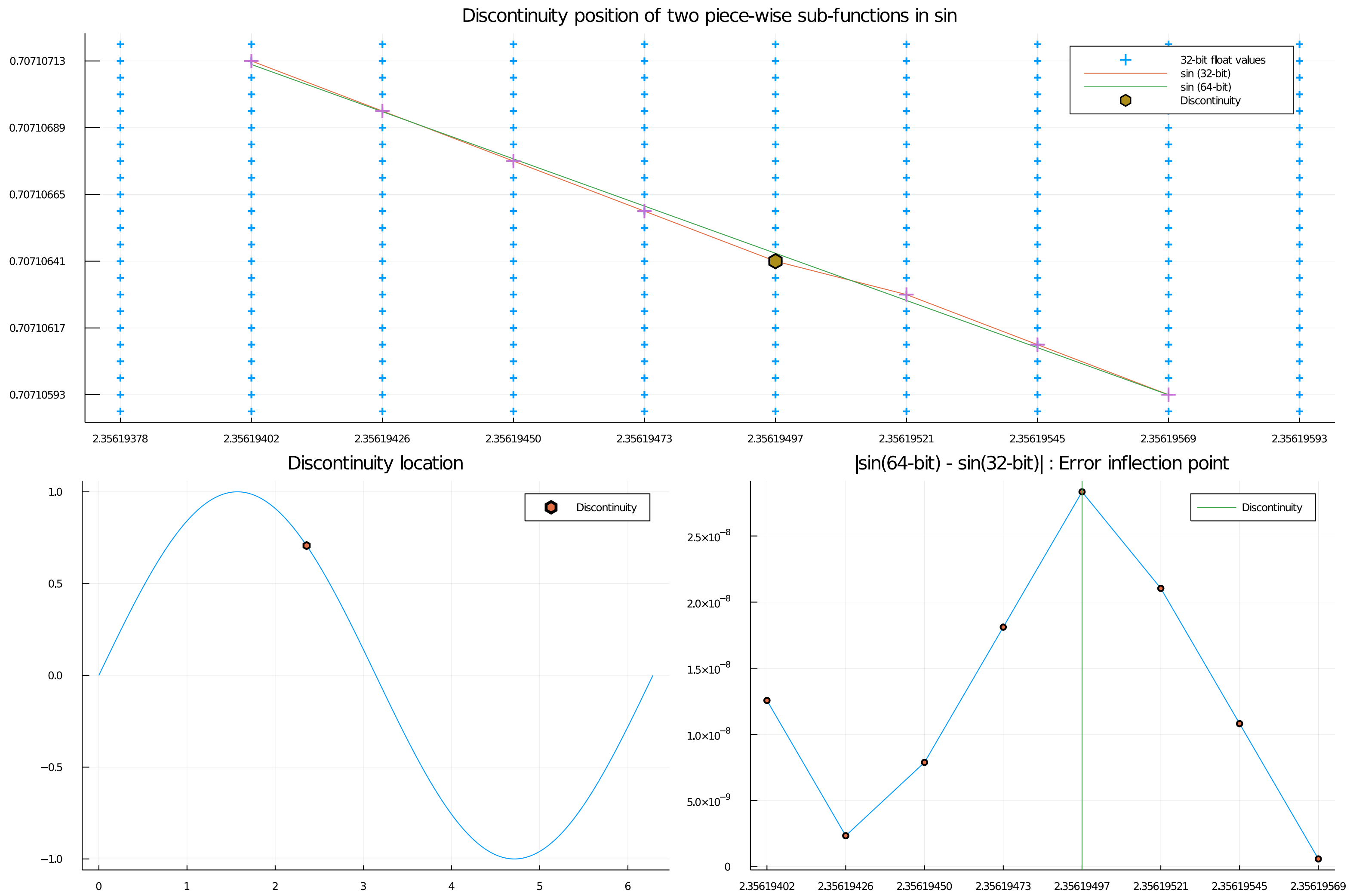


Figure 3.2: A close analysis of the values at one of the `sin` discontinuities and the associated error with each.

If you are unfamiliar with how interpolation is used to improve the accuracy of the function being approximated, see Appendix Section A.1.1. In that appendix an approximation of `sin` is shown to different levels of precision using different techniques.

3.2.4 Polynomial Approximation

Creating approximations using polynomials is an alternative to tables when computations are cheap and memory lookups are expensive. It is also possible to use them in conjunction with tables (similar to the example with gradient interpolation shown in Appendix Section A.1.1.3).

The polynomial approaches can generally be broken into a few different classes. Those which are mathematically proven to converge on the correct answer (Infinite Series, Continuous Fractions), rough approximations (basic Taylor series/rational function fitting) and error minimising approaches (Chebychev/Remez).

The mathematically proven solutions require a large number of precise terms to converge correctly, and many identities quickly become limited by the precision of the floating-point representations.

Taylor series approximations can be suitable for small sections of a curve but the error they exhibit can be extreme and not equally distributed.

Error minimisation polynomials, such as orthonormal polynomials, are an improvement over basic Taylor series approaches as the error exhibited is more evenly distributed. Meaning, if the fit of the function exhibits a worst-case error smaller than what is representable in the final output precision then you will have a correctly rounded polynomial fit.

In this dissertation I rely on these approaches when looking into how to generate *good-enough* approximations, particularly in Chapters 6 and 7. If you are unfamiliar with these approaches and how they are applied to approximate a function for a given range of inputs, Appendix Section A.1.2 gives a thorough demonstration and explanation for many different approaches used as well as a brief outline of the performance and accuracy implementations of each.

3.3 Implementing Elementary Functions

For many programmers, the elementary function implementations provided by their mathematics package may not be something they ever think about. But, to fully understand how it is possible to modify or improve these functions it is necessary to know and understand how they are currently implemented.

The implementations require specific approaches to achieve a high level of accuracy and the expected behaviour. To cover this I will go through the implementations of a number of the functions using the Julia mathematical functions as a reference.

3.3.1 Implementing `sin` & `cos`

The `sin` function is the cos function shifted by $-\frac{\pi}{2}$.

$$sin(x) = cos(x - \frac{\pi}{2}) \tag{3.1}$$

As a result of their similarity, the two functions can be implemented similarly. One might implement one and then when calling the other simply offset the input. However, this is likely to result in incorrect rounding as either the input value may be in a less precise binade in floating-point or the offset may cause rounding of the input and therefore produce the result for the wrong input. Because of this, each function is still implemented as its own method, but the functions look very similar.

Alternatively, you might be familiar with the Taylor function representation of Sine and Cosine (see the earlier Section 3.2, which covers basic approximations):

$$\begin{aligned} sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... \\ cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... \end{aligned}$$

Besides the high cost of these functions (which may be somewhat mitigated with correct constant expression evaluation), it is difficult to determine the absolute number of steps necessary to give the precision needed for all potential inputs without having to have an arbitrarily large number of steps. Figure A.4a shows how even with a relatively high number of terms accuracy is still relatively low and quickly degrades for large input. This could be mitigated somewhat by efficient range-reduction to the $[0, 2\pi]$ range, but this would still not be correctly rounded without further work.

To simplify this, the general approach to writing these functions is to break the function into parts. First, consider special cases for inputs near zero. For `cos` where inputs are close to zero the function returns one. For `sin`, for inputs very close to zero the function will return the input. These early out steps work due to the very close to zero and very close to one gradient when at the start of `cos` and `sin` respectively. With these gradients and the low precision of 32-bit floating-point numbers, there is not enough precision for there to be a difference. These cut-off values for an early out can be empirically tested.

If the input value is not close to zero, then more work is needed. `sin` and `cos` are repeating periodic functions which can be represented by as a piece-wise curve of two distinct sub-functions. Those sub-functions being *(a)*, the close to one gradient as the functions makes its most direct path from minus one to one, and *(b)*, the rounded caps as the curve changes direction. Each sub-function, *(a)* and *(b)* only need to be computed for the first half of their domain as they are also symmetric. The sub-functions and how they are used to construct the full curve are shown and covered in more detail later in Chapter 6 and shown in Figure 6.2.

Both of these half-sections can be represented by a polynomial very accurately (see Section 3.2). With this knowledge, the problem then becomes one of selecting the appropriate section of the curve based on the input when calling `sin` or `cos`. This is achieved with the range-reduction step of the algorithm. The input is divided by $\frac{\pi}{2}$ and the quadrant of the range, as well as the remainder, is returned. The quadrant value tells us which section of the curve the input value lies in, and the remainder is the input to the reduced section sub-function.

If the range-reduction and polynomial representation of the sections are "good-enough" then this will result in a good `sin` and `cos` function.

3.3.2 Implementing `asin`

The inverse sine function, `asin`, is flipped and symmetric around $x = 0.0$. This means to implement the function, only $asin(|x|)$ is required to be computed. `asin` where $x < 0.0$ is computed by calling $asin(|x|)$ and flipping the sign of the result.

In the early years of numerical computing a formula was constructed based on Chebychev polynomials to approximate the $0 \leq x \leq 1$ range [80]. That formula is:

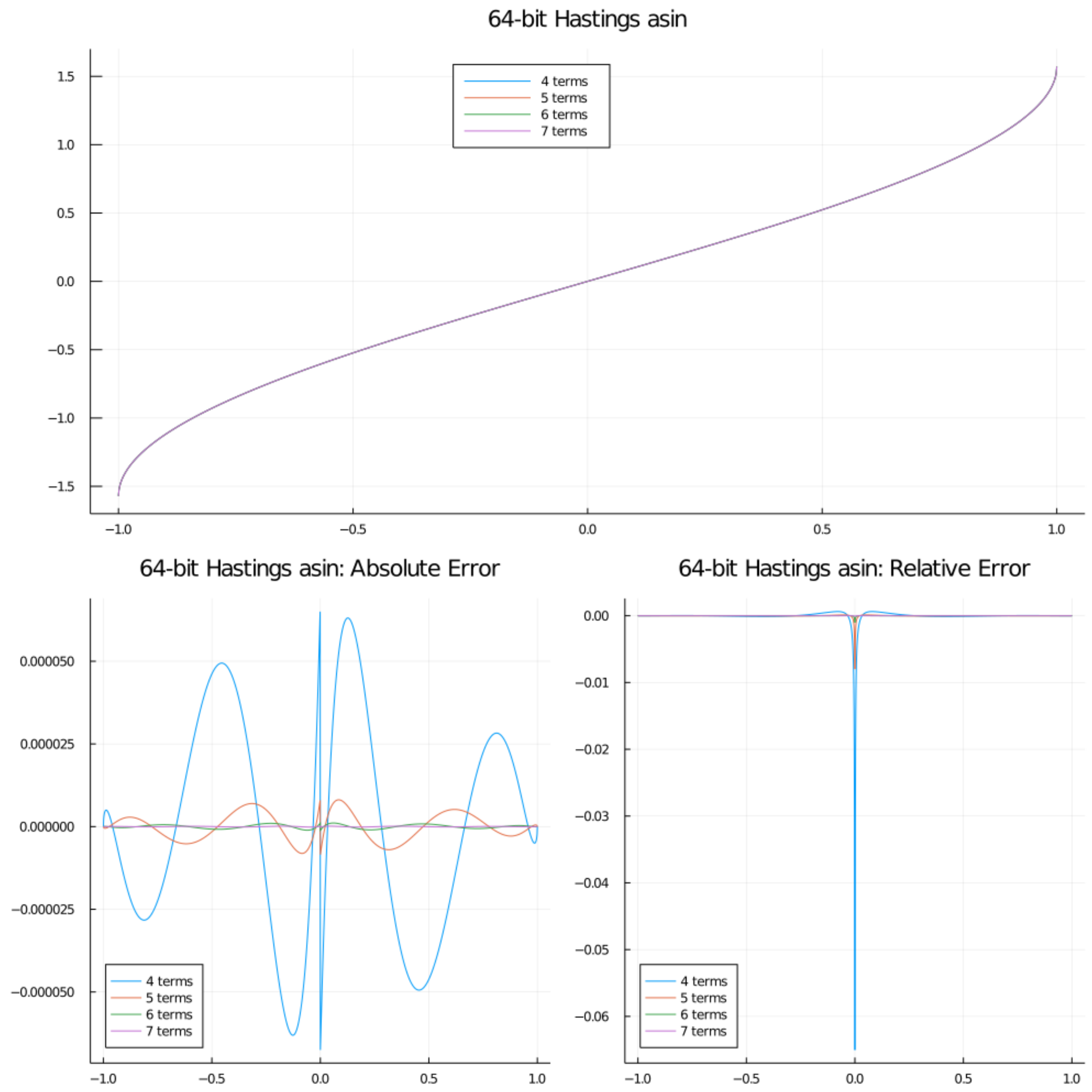
$$\text{asin}(|x|) = \frac{x}{2} - \sqrt{1-x} * (a_0 + a_1 x + a_2 x^2 \ldots a_n x^n)$$

Example constant values for the 4 term implementation:

$$a_0 = 1.5707288$$
$$a_1 = -0.2121144$$
$$a_2 = 0.0742610$$
$$a_3 = -0.0187293$$

This gives the result shown in Figure 3.3 and these maximum errors when using the constants from the original Princeton University monograph [80][8]:

[8] While history is not the focus of this dissertation, I find it is important to bring to the attention to the reader the significance of this particular document. The simple algorithmic approximations, derived constants and magic numbers found in this paper are incredibly influential and turn up in mathematical textbooks, public source-code and mathematical libraries for decades after its publication. This is quite remarkable given the relatively low precision of the constants and very simple algorithms provided.



Terms	**Max. Absolute Error**
4	6.7678×10^{-5}
5	8.5521×10^{-6}
6	1.3411×10^{-6}
7	5.3644×10^{-7}

Figure 3.3: An implementation of the `asin` function as described by Hastings et al. in the Princeton University 1955 publication on the implementation of mathematical functions on digital computers.

This gives a relatively small number of operations to construct a very good (but not correctly rounded) approximation of `asin`. The downsides of this approach are the inaccuracy and the reliance on a good existing `sqrt` functions.

The modern implementations of this function use more steps to try and increase accuracy. Firstly, as this is `asin` the function $asin(x) == x$ can be used when x is small. This is the same trick used by `sin`.

Next, the $asin(|x|)$ can split the curve in half, so that there is a relatively straight part where $x < 0.5$ and then a curved section where $x > 0.5$.

The straighter section, $|x| \in [0.0, 0.5]$, uses a simple rational polynomial to approximate the curve. For 32-bit floating-point, the curved section uses a modified version of the formula used in the simple Princeton University example except the Chebychev polynomial is replaced with a rational polynomial using more terms. This provides enough accuracy in 32-bit. For the 64-bit implementation, the curve is split again before the point of highest change of gradient and a different polynomial is used for each of the parts of the curved section.

When these piece-wise functions are placed back together the result is much higher accuracy than the simple direct implementation. The difference in the error when using 32-bit can be seen in Figure 3.4 where both implementations are compared to the correctly rounded solution. While the more in-depth modern implementation provides more accurate results for most inputs, it is still interesting to see that for a few select inputs the simple Chebychev direct polynomial can still produce a better result.

3.4 Function Equality

In this section we introduce different views on equality of functions, define how we will be assessing functional equality for use with approximate computing theory and showing where these definitions of equality matter in industry applications.

Defining how we understand equality is important for enabling safe, sensible approximate computing approaches. Without it we would not be able to replace one procedure with another and guarantee an *equal* result. There would be no metrics available to prove that the new procedure is equal to the original in a meaningful way.

My approach in this section is to begin with defining equality of the state of the input and output to a function, definition of functions themselves and then move onto definitions which rely on the state of the inputs and outputs of objects using those functions. We will then show how each of these different definitions for equality work together and how they can be applied in the field of approximate computing.

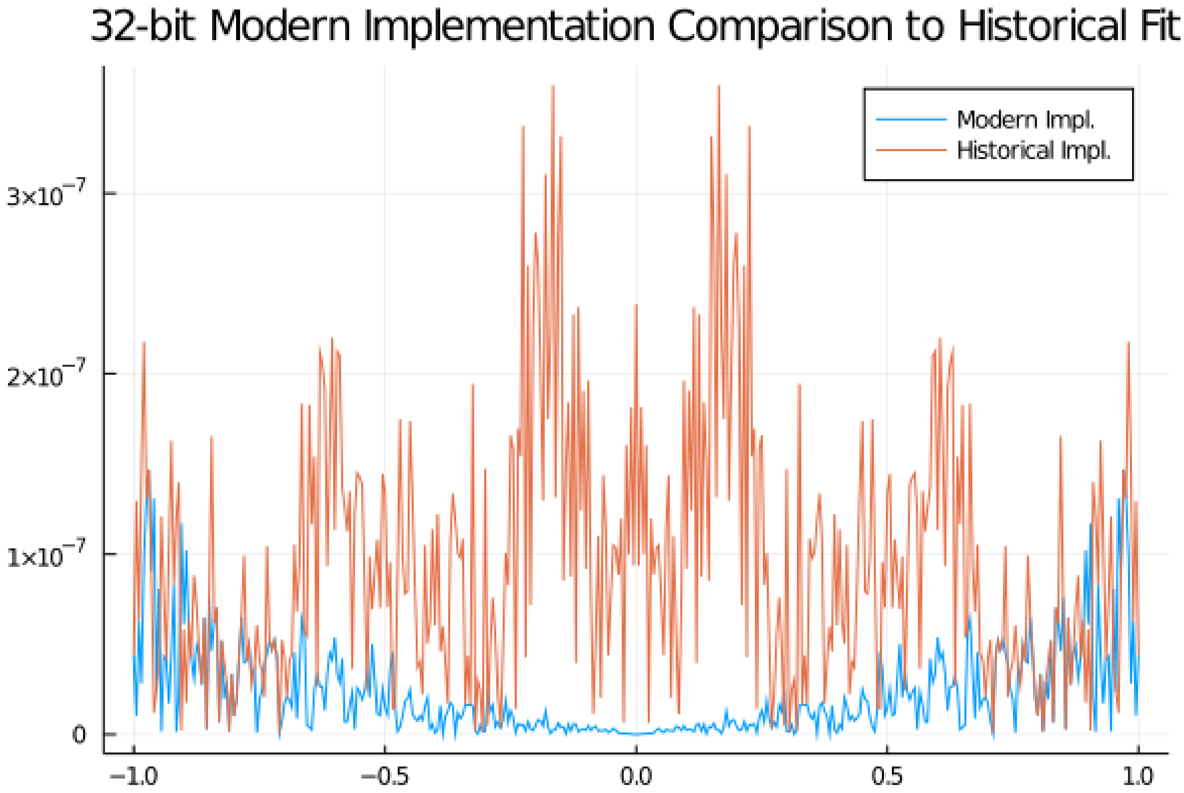


Figure 3.4: A comparison of absolute error from the correctly rounded results of the current `asin` implementation in the Julia programming language as it compares to the seven term version of the 1955 Princeton University function definition.

3.4.1 Internal Vs External Equality

During software development it is common to refactor functions, change or update libraries and to provide multiple type interfaces to a single function. Looking at code from a high-level this allows for the programmer calling function $f(X :: TypeT)$ with known pre- and post-conditions to assume that the function being called is the one expected even though the exact implementation of function f may be different in the underlying implementation based on how $TypeT$ is defined.

In type theory, treating all implementations of a function with the same interface as the same function is known as extensional equality. That is, implementations of f as f_1 and f_2 are considered the same function if from the calling position they exhibit the same name and type inputs, even if the underlying implementation differs.

Working with complex codebases relies on extensional equality so that different implementations of the same function can be used on different systems but still be confident that the functions will perform the same action, even if the action is not performed with an identical set of instructions. One of example of this might be compiling against a different implementation of a standard library. Each implementation should be following the same standard and providing the same interface and therefore should behave as the user expects. It also allows for programmers to refactor functions internally to provide better performance on some systems or unlock functionality specific to a platform without having to change the way the code is accessed at the higher level where to the end-user the function is *equal*.

Fundamentally, this is also why an optimising compiler is able to perform its optimisation pass transforms in a stable manner. It changes the internal structure of the code to be more optimal while guaranteeing the same rules that the programmer requires at the higher level.

For extensional equality to be applied the rules for it must be defined, as different use-cases may require different rules. For example, $f(x :: int32)$ may be considered extensionally equal to $f(x :: size\_t)$ when $size\_t$ is equal to $int32$, or if there is a valid implicit constructor for $int32$ from $size\_t$. The definition of extensional equality is therefore coarse [158] and needs to be fully defined for each use-case before actions can be taken to change the code.

The opposite of extensional equality is intensional equality. For two functions to be intensionally equal they must be exactly equal in their implementation as well as their definition. This means that a function defined by a standard library which is implemented in two separate libraries with the same guidelines and rules would not be considered equal unless the source-code (or compiled object) were identical.

This is a much stricter definition of equality. If this standard of equality was to be enforced then applications would require specific implementations and dynamic linking to alternative implementations would not be possible.

However, this type of equality testing is still in use for some software, in particular cryptographically signed applications [40] and files and software used in piracy protections [26].

```
int f(int x)
{
   int sum = 0;
   for(int i = 0; i < x; i++)
      sum += 2;
   return sum;
}

int f(int x)
{
  return 2 * x;
}

```

Figure 3.5: Example of extensionally equal implementations

An example of this for software is the video-game “MOTHER 2 / EarthBound” which implemented its own anti-piracy DRM (Digital Rights Management) using checksums of machine code instructions. By producing a checksum of a specific range of instructions they attempt to ensure that the functions at those addresses are intrinsically equal to what was originally assigned to the disk. If the checksum values do not match, the video game is able to determine that the instructions may have been altered in a way to circumnavigate copy protection or to enable unspecified behaviour (such as cheating). As a result, the game will then not work as intended and results in a detrimental play experience for the user.

This summarises the two types of equality defined by the state of the function implementation and definition that we are interested in. One where the user is agnostic to the state of the internal working as long as the function is conforming to some defined standard of behaviour, and another where the user requires absolute identical instructions at a specific level.

3.4.2 Input and Output Equality

When covering extensional equality it was mentioned that the input and outputs of a function would conform to a standard so that, to the user of the function, the internal state did not matter. That is to say for any implementation of $f_{0...n}$, which followed a specification defining how f behaves, then an input of x should always return a y conforming to the standard regardless of the underlying implementation. Also, all input values which are valid for f_0 will also be valid for f_1, no valid value input to f_0 should fail on f_1.

If we were to consider only the inputs and outputs of a function when judging the equality of two functions we would be interested in three things:

- First, the **Domain** of function f. This is all the valid inputs to function f.
- Secondly, the **Co-domain** of f. This is all the valid outputs of the function f

- And finally, the **Range** of f. This is all the valid outputs of f which are are a result of the inputs from the **Domain**.

You will note that the **Co-domain** and **Range** are very similar concepts. However, it is important to distinguish all possible results of a function from those which are valid within the **Co-Domain** as some functions may be defined to only be valid within specific boundaries. For example, a function which takes a temperature in Kelvin and converts it to Celsius will not accept values below zero as input even though the instructions may not fail when processing it, therefore no output values would exist in the Range for input values below zero but they would exist in the co-domain.

With these three key identifiers we can begin to define equality for given functions based on input-output pairs. In this context an implementation of a function is considered equal to another implementation, regardless of method or definition, if the input domain and the range are identical. If the co-domain of the function did not match this would not be a problem as the function may be undefined at the input positions which would result in some values which may be contained within the co-domain.

In practice, a specification or standard will often define the valid input ranges for a given function and the expected behaviour for each which would produce the range. The functions then implemented have a clear definition to use to test if they conform and are *equal* to function defined by the specification.

In pure mathematics this approach is very simple and easy to define. When this approach is then applied to software development however, the definitions can become somewhat complicated by the nature of numerical representations on a computer with finite bits to correctly represent values.

Starting with an uncomplicated example, the function $f(x) = x + 1$ may be defined for inputs $0 \rightarrow \infty$. Mathematically, this is fine, we can easily define the range to be $1 \rightarrow \infty + 1$ and then any other implementations of f which have the same domain and range will be valid equal functions by our current definition of equality as they would be identical.

Unfortunately, on a computer this is not so easy. We must assign some way of representing the input x in hardware so that we may perform the necessary computation. If the input is represented an 8-bit unsigned integer then we have a problem — we are only able to create a function with a domain of 0-255 and a range of [[1-255], 0]. This is wrong for two main reasons, firstly we do not match the domain as we are only able to represent inputs up to an integer value of 255 and secondly we produce an error as an input of 255 will return 0, not 256, so we are not even equal to the original function f for the reduced domain of [0-255].

This is obviously a trite example as any decent specification would define an integer input between the largest and smallest representable values of the data-type, but it works as an example of the kind of differences in domain, co-domain and range we need to consider next: dealing with finite representations of real numbers when considering equality.

Floating-point is the most widespread representation of *real* numbers used in computers today [119]. Specifically, most hardware implements floating-point numbers by the *IEEE-754 Standard for Binary Floating-Point Arithmetic* [3]. A more detailed explanation for how this standard works and affects the accuracy and performance of functions can be found in Chapter 4 or alternatively a detailed explanation outside the context of approximation can be found here [6]. For this section, we can summarise the main limitations of IEEE floating-point that we care about as:

3.4.2.1 Limited Bits

The number of bits which can be used to represent the number are limited which results in an inability to correctly represent all possible numbers. This can be resolved with variable length encoded floating point [163] but this is not commonly available in hardware [19] and is very costly to implement in software making it practically unusable for some performance critical applications.

3.4.2.2 Accuracy Degradation

Due to IEEE floating-point numbers being based on an exponential curve, as the number to represent moves away from zero the distance between each representable number becomes larger. As a result, calculations involving larger numbers may cause less accurate results. This makes comparing the equality of a theoretical implementation with the floating-point implementation difficult as the result may diverge due to rounding to a representable value at many stages through the implementation. Where this rounding takes place can often be dependent on specific hardware or software implementations which vary in how calculations are handled.

3.4.2.3 Mixed-Precision Results

The IEEE-754 standard gives certain guarantees about the result of basic arithmetic when using floating-point numbers. However, due to different hardware implementations the same code may produce different results on different platforms depending on where the rounding takes place [127]. This type of error is caused when the numbers being calculated are held in a larger precision registers to perform the calculation. The precision of that register may result in a different value being found to be the result of the calculation which may then round to a different result when it is passed back into the lower-precision register.

Further differences can occur if the computing device supports SIMD or fused multiply-add (FMA) instructions. As each of these approaches can result in different outputs depending on how they are implemented.

SIMD may cause different results by using different size intermediate registers, or performing its rounding at different stages.

Fused multiply-add keeps the results in the higher precision registers across two operations, effectively removing one possible incorrect rounding step. This results in more accurate answers which are therefore not equal to those produced by simply performing a multiple and then an addition through regular floating-point arithmetic instructions. This discrepancy is actually one of the benefits of this optimised instruction, and is used in numerous algorithms, such as optimised *range-reduction*, to produce improved results [118].

This means that SIMD and FMA, which may be valid transforms when we are only ensuring extensional equality to a lenient standard and would still be considered *equal*, would result in different answers when we compare the implementations based on the domain, co-domain and range.

With these three limitations we can show that functions even as simple as $f(m,x,c) = mx + c$ could give different results depending on how it is compiled and what hardware is being used. This also shows that as the values for m, x and c move away from zero, the error in rounding is also more likely to occur. As a result, two functions written in a high-level language and compiled may not share domain, co-domain and range values and under the form of input/output equality would not be considered truly equal, while at the same time appearing extensionally equal.

3.4.3 Function Transforms

So far, the co-domain has not proven too useful in identifying function equality. The co-domain becomes useful when we start to consider function transforms. Remembering back to studying basic algebra, the function $f(x) = \frac{x^2}{x}$ can be simplified to $f(x) = x$. This is fairly obvious and we could say that the functions would be equal. So lets take these two implementations and compare them for a short range of inputs:

Input	$f(x) = \frac{x^2}{x}$	$f(x) = x$
-2	-2	-2
-1	-1	-1
0	NaN	0
1	1	1
2	2	2

And there is the problem! These two implementations which by basic mathematical logic should be thought of as equal are here shown to be not equal for the input value of zero when processed on a computer, as we are unable to divide 0^2 by zero. For examples like this we say that the functions are not equal because there exists a value in the co-domain which is not shared between both implementations. These differences may occur from simple mathematical rules like this but can also be caused by transforms which would not be valid under IEEE-754 floating-point rules.

This is a problem in the co-domain and not the range because the input value of zero may not be in the domain, but in the event of an incorrect input it would result in an exception that the user would not expect. These types of errors of equality can be catastrophic in some cases [125].

3.4.4 Performance in Input-Output Equality

For some classes of programs, such as security and cryptography, where input-output equality is a necessity, performance is also a metric for equality. To be secure, many functions are required to finish in the same amount of time for the same inputs.

As an example, consider the well known timing attack example of a string comparison function.[9] It is possible to return false as soon as a single character is identified to not be the same. By returning out of the function early some performance has been saved but it has made the system vulnerable to a timing attack [22] by revealing information regarding the content of the string being compared by how quickly the function returns.

As a result, despite the same input-output pairs a function which might return early or in other ways perform a different number of actions depending on the input function will reveal information about the input or the methods used to process it. Therefore these functions, in terms of results, are extensionally equal. But provide extra information due to intensional inequality in how data is processed.

3.4.5 Intersection of Equalities

What has been explained in the preceding sections can be demonstrated through a list of simple examples to show how these two types of equality interact.

Extensionally Equal and I/O Equal

Implementation A: $f(x :: Int) = x + x + x + x$

Implementation B: $f(x :: Int) = 4 * x$

As integers do not incur loss on addition without valid input ranges. This function exposes two equal definitions with differing internals which do not result in a different input-output pairs.

Extensionally Equal and I/O Not Equal

Implementation A: $f(x :: Float) = x + x + x + x$

[9] This style of timing attack and similar examples were made popular by Paul Kocher with his cryptography research demonstrating the weakness when comparing RSA tokens [94].

Implementation B: $f(x :: Float) = 4.0 * x$

In floating-point, $x + x + x + x$ is not equal to $4x$ as each addition may result in rounding which could impact the final answer. As a result, this implementation is extensionally equal (if the function has a tolerant standard) but not I/O Equal.

Intensionally Equal and I/O Equal

On Intel x87 80-bit extended precision without FMA operations
Implementation A: $f(x :: Float) = 2.0 * x + 1.0$
On Intel x87 80-bit extended precision without FMA operations
Implementation B: $f(x :: Float) = 2.0 * x + 1.0$

Both of these implementations are intensionally the same, that is to say they have the same implementation and definition. They are also running on the same platform. Therefore they will result in the same input-output pairs.

Intensionally Equal and I/O Not Equal

On Intel x87 80-bit extended precision without FMA operations
Implementation A: $f(x :: Float) = 2.0 * x + 1.0$
On Intel AVX 128-bit extended precision with FMA operations
Implementation B: $f(x :: Float) = 2.0 * x + 1.0$

This is the same implementation as above, but running on different platforms. Due to the different extended precision used on each platform this calculation may have different outputs on the different machines. The output may also differ if an optimised compiler is used which converts the multiply and add to a fused multiply-add operation. Both methods would change how or where rounding takes place and cause a mismatch in input-output pairs.

Generally, these changes may not result in a change of value that the user would find significant, but it is important to consider these inequalities when looking at understanding how error is tolerated and where it can be exploited based on what is important to the user.

3.4.6 Application to Approximate Computing Theory

This chapter has given a high-level overview of different ways of considering equality of functions and shown some simple examples of how these apply in different combinations with software implementations to produce scenarios with output non-equal results while still being considered equal in some way. I now want to elucidate how this all applies to the field of approximate computation and how these known and accepted inequalities can be exploited to produce more efficient but still *equal* functions.

In software engineering, the error from functions is more common than a lot of programmers may realise as the average programmer may not consider the full stack of language, compiler, optimiser and hardware when writing a program - sometimes to the point of resulting in death [8]!

Despite this many programs exist with significant error without any catastrophic consequences. Intuitively this tells us that the programs which exist with error must be exhibiting error within a tolerable range and this is key when thinking of approximating a function.

Generally, a program that exhibits error but does not cause a crash is not designed to exhibit that error. This means that the program is likely aiming to use the most accurate methods possible to reach its result. So, let us format this in terms of the equality that have been discussed so far.

For most applications we encounter, we do not have the full information for the use-case of that function or knowledge of everywhere which it may be used. As such, the best information usually available is the error that is exhibited in existing implementations.

Given an implementation that exhibits some error that is being used in an application, we will call this $Impl_{Error}(x :: TypeT)$ that is an implementation of the function $Impl$ with no error which we can call $Impl_{Precise}(x :: TypeT)$. The implementations differ but share the same definition, so that makes them extensionally equal. They also share an input domain, but they do not share the same output range, so they are not I/O equal.

This tells us that, in the program in which this implementation exists, it is acceptable for the function $Impl$ to exhibit, at least, up to the error of $Impl_{error}$. If we want to quantify that error, we may consider the average error, min-max error or the individual error per input depending on the context of the program (see Section A.3) by comparing the function to a more precise version, in this case, $Impl_{precise}$. For simplicity here, we will consider the maximum error of the set of the whole range of $Impl_E rror$. This tells us that this function gives a maximum error of:

$$\begin{aligned} x &= [...] \\ err_{range} &= \max\nolimits_{1\leq i\leq N}\left(\mathrm{Impl}_{\mathrm{Precise}}(x_i) - \mathrm{Impl}_{\mathrm{Error}}(x_i)\right) \end{aligned}$$

So this shows that the function being implemented supports *at least* this level of error.

Therefore, if we wanted to create a faster implementation of the $Impl_{Error}$ function we would choose to focus on making the implementation as fast as possible within the constraints of the error limit that $Impl_{Error}$ currently exhibits.

Any function that could be implemented which results in an error less than or equal to err_{range} at a faster run-time performance would be an improvement to the program and by our equality constraints, it would be considered *equal* and therefore not negatively impact the program.

Taking this further, we could explore the true *equality limit*. That is, to test where the true maximum acceptable error exists. If the full context and use-cases of the function are known, this can be explored by increasing the difference in the range between $\text{Impl}_{Precise}$ and a new implementation until the program no longer performs satisfactorily. This may give a much larger optimisation space to work with without violating the extensional equality and gives a better limit to the I/O equality. This approach is demonstrated in Chapter 6.

As most programs are not written with the maximum acceptable error taken into account, our defined rules for equality allow us to exploit mathematical functions to find a function which better fits with the true requirements of the program. This area of computing is sometimes known as *good-enough* computing, where the target of the program is to do only the required work and nothing more.

3.4.7 Summary

In this section, the different ways we consider functions to be equal has been defined, which guides how it is appropriate to modify existing functions within a program when looking to apply approximate computing optimisations. This outline can be used for reference in how the impact of approximations as function replacements are considered in the rest of this dissertation.

3.5 Conclusion

This chapter has provided a detailed overview of the specific work and ideas introduced in Chapter 2. I have covered in detail the history of the numerical systems used for mathematical functions, how they are defined, how I will measure error and what restrictions it places on the development of these functions.

Different mathematical approaches which are commonly used in the development of mathematical functions are then covered. This is necessary to understand for when I will be changing and modifying them so that there is an understanding of the performance and accuracy trade-offs being made. Examples of how mathematical functions are commonly implemented are then presented, showing the strengths and weaknesses of different approaches.

This is concluded by giving a detailed background to what it means to be *equal* in software and how this effects approximation and even non-approximation approaches to editing and improving mathematical function implementations.

This should provide enough grounding for the chapters which follow where I will be applying approximation techniques and discussing where correct answers are important and where the correct answers are not important.

Chapter 4

Evaluating IEEE-754 Elementary Functions

As was shown in the Chapter 3, accurate and reliable elementary functions are important to the correctness and portability of mathematical software and current implementations are guided by the *IEEE-754* standard for floating-point numbers. That standard provides a list of recommended functions, which are a subset of elementary functions. The latest version of this standard comes with a list of recommended elementary functions to be implemented and guidelines on how they should behave. This includes: the function input domain, the types of exception which should be produced, and a requirement that the returned result is correctly rounded for the selected rounding-mode. The list of single-input functions which are recommended by *IEEE-754* in Table 4.1.[1]

For some elementary functions, absolute correct-rounding can be very difficult to achieve with good performance due to what is known as the *Table-maker's Dilemma* [102] which shows it is impossible to know how many digits of precision are needed in advance.

In this chapter, function accuracy will be measured using *Units-in-Last-Place* (ULP). This is the measure of the distance between two floating-point numbers (see Section 3.1.3). The result of a calculation is correctly rounded when the ULP error is less than or equal to 0.5 (when in Round-to-Nearest rounding mode).

The current standard mathematical libraries used in Clang, glibc, Microsoft C Runtime and Julia are shown to not be correctly rounded and do not follow the definitions of mathematical functions, which require correct-rounding, in the IEEE-754-2019 standard, with the majority of the functions being in the 1 to 2 ULP range and two functions with an error of 3 ULP.

Many practitioners consider a 1 or 2 ULP error in a floating-point number to be irrelevant as the mathematics of floating-point numbers means any arithmetic using the result may be likely to incur this error anyway. In some cases, this is a fair assumption, but in many other cases, it is not as some algorithms require bit-precise answers. The problem is compounded because

[1] Excluding the helper functions for working in degrees instead of radians, such as `sinPi`, `cosPi`, and `tanPi`.

not only are these functions incorrect, but they are incorrect for different input values between implementations. This means that the results are not consistent between libraries, and any further mathematics on a result may cause different compounding errors. As the absolute error is small, this is unlikely to result in a significant change of result in most applications, but it does still affect many which rely on bit-perfect results.

Consider the computer graphics imagery used in many modern movies. In the rendering process, many computers are used in parallel to calculate the final frames that are composited into the final video. Each frame requires a large amount of texture data for the 3D assets and some parts of those assets are procedurally generated (think of scratch patterns and dirt textures) to reduce the memory impact across the many computers. We might distribute frame one to computer A which dynamically links to library X and frame two to computer B which dynamically links to library Y. If those two libraries generate different patterns and then we stitch the rendered images together in the final movie, we would see a flickering pattern between the two different results.

This is exactly the type of portability problem which the IEEE-754 standard was introduced to prevent, and it has for arithmetic, but for the recommended functions it seems to have been ignored possibly for performance gains or because the impact on the majority of programs is considered by the community to be negligible. Compliance with the standard is important for allowing the production and sharing of stable and portable software. Not following the standard moves the responsibility of testing onto the programmer who may be unaware that testing is required and that could result in broken software being released unknowingly.

A thorough breakdown of the IEEE-754 standard and the history and motivations for its development can be found in Chapter 3. As covered in that chapter, it is noted that the later versions of the standard (2008 and 2019) are more strict with the requirements for recommended functions and in this chapter, results are presented which show that it appears common mathematics libraries have not kept up as many floating-point mathematical libraries are not complying with the requirement to be correctly rounded.

My initial investigation into this, upon discovering that the standard may not be currently adhered to, led me to speak to some maintainers of these projects who told me that the reasons for not complying with the standard were down to performance constraints. I did not believe the performance to be a limitation to compliance and believe that either we should seek out a correctly rounded implementation that has desirable performance characteristics or we should provide evidence for the IEEE committee so that the standard can be changed to reflect the actual performance constrained state of the industry. In particular, I want to ensure that the results from different libraries provided the same outputs for the same inputs so that the portability benefits of the standard can be safely maintained.

In this chapter, the current behaviour of 32-bit single-input mathematical functions in three modern CPU implementations are analysed, one older pre-2008 implementation and one popular

Function	Domain
sin	$(-\infty,\infty)$
cos	
tan	
asin	$[-1,+1]$
acos	
atan	$[-\infty,\infty]$
sinh	
cosh	
tanh	
asinh	
acosh	$[+1,\infty]$
atanh	$[-1,+1]$

Function	Domain
exp	$[-\infty,\infty]$
expm1	
exp2	
exp2m1	
exp10	
exp10m1	
log	$[0,\infty]$
log2	
log10	
rSqrt	
logp1	$[-1,\infty]$
log2p1	
log10p1	

Table 4.1: All single input IEEE-754 recommended functions, excluding the functions which take degrees as input.

GPU implementation. After the current implementations which produce incorrect-rounding have been shown, three distinct use-cases where incorrect rounding results in non-portable code is presented: direct mathematics, procedural content generation and a convolutional neural network. This work is followed with a simple to implement correctly rounded solution for 32-bit floating-point elementary function in Chapter. 5, and an implementation which exploits the one ULP max-error bound implied by these libraries in Chapter 6.

4.1 The Recommended Functions

Table 4.1 lists all the functions specified by the IEEE-754 standard along with the valid ranges of input for each. These functions were first specified in the 2008 revision of the standard [2]. Before that, the 1985 version only mentions the elementary functions in the foreword, which states that one of the guiding reasons for the development of the standard is to "Provide for the development of . . . standard elementary functions such as exp and cos".

The introduction of the list of recommended functions in the 2008 revision are in Section "9.2 Recommended correctly rounded functions" with the requirements that "A conforming function shall return results correctly rounded for the applicable rounding direction for all operands in its domain". For this work, only the single-input functions are considered because the domain of the functions taking two inputs is larger than it is possible to test every possible input in a reasonable time. This is also the reason only the 32-bit implementations are considered and not the 64-bit implementations. Advances in computer performance mean that it is possible to evaluate every possible input of a single-input 32-bit function in a few hours. I have also found examples of incorrect rounding in 64-bit but it is not possible to comprehensively test all possible inputs as the sheer number of values would mean it would take an infeasible amount of time.

32-bit floating-point is widely used where the performance or memory usage is a major consideration. It is common in legacy software, GPGPU (General Purpose GPU processing), machine learning and is the default size of the `float` type in C and C++ [84, 85].

4.2 Libraries Evaluated

The following libraries are used and analysed for understanding approximation and conformance with IEEE-754:

- Julia
- MCRT (Microsoft C Runtime Library)
- gnulibc
- FDLIBM

4.2.1 Julia

Julia is a relatively new programming language aiming to be used for data science and other technical mathematical use-cases. Its goal of being used for mathematics is clear even in its syntax which allows for writing code that is visually similar to formal mathematical equations. Interest in this library is due to many of its mathematical function implementations being newly written (post-IEEE-754) and not simply a reuse of code from older libraries.

Originally, Julia was linking to OpenLibm which itself is evolved from glibc with some updates. However, one of Julia's advantages over other static languages is the JIT (Just-in-Time) compiler and other compiler features which allow for function specialisation. As a result, it is desirable for the language to implement its mathematical functions directly in Julia so they can gain a performance benefit from the JIT at runtime. This means that any implementations which are not a direct translation into Julia have been implemented with the current knowledge and goals of the functions in mind. Unlike other libraries which can claim 'legacy' implementations as a reason for non-conformance with IEEE-754, Julia's conformance or non-conformance will be deliberate as the functions are written *after* the new requirements of the IEEE-754 standard.

While Julia is a relatively new language, it has a large user base and is beginning to see wider use. Notably, Julia now provided as a pre-packaged container on the NVIDIA GPU Cloud platform and as of 2020, it is seeing real-world use at NASA.

4.2.2 Microsoft C Runtime Library

The Microsoft C Runtime Library is provided with all Microsoft Windows installations. For the latest Windows release, that represents over 1 billion installations. Unlike other libraries being analysed this one is closed source.

This library is being used for analysis of current approximations because of its widespread adoption and use.

4.2.3 glibc

GNU Lib C is the standard C and mathematical library that is supplied with a number of distributions of Linux. Particularly, it is the standard library provided with the popular Ubuntu and Debian Linux distributions.

This implementation was chosen as a counter-point to Microsoft CRT because it also has wide scale usage. This library is open source which allows us to understand the behaviour easier than the Microsoft CRT counterpart.

4.3 Accuracy of 32-bit Functions

Each 32-bit single-input function was brute-force evaluated for all valid 32-bit floating-point inputs and compared against the correctly rounded result from GNU MPFR[172]. The worst case ULP error and the total number of input values that yield an incorrect answer were recorded.

In Table 4.2 the results are shown and colour coded in the ULP column to highlight how compliant each implementation is with the IEEE-754 standard, and more subjectively highlighted in the 'Incorrect (%)' column to mark the frequency of mismatches with 0% being highlighted in green, $< 1\%$ in yellow, $< 10\%$ in orange and any larger values in red. In the ULP columns, the majority are 1 ULP. There are a surprising fourteen instances of 2 ULP functions as well as a few larger errors exclusive to the Microsoft CRT library.

Note that the `exp10` function is correctly rounded in glibc but not correctly rounded in Julia. Both have a history of implementations which are taken from FDLibm and the FDLibm implementation of `exp10` is also correctly rounded. For example, glibc's `tan` function appears to be identical to the FDLibm implementation and Julia is based on OpenLibm which itself is related to FDLibm. In this test, the FDLibm and glibc `exp10` show the same error distribution, ULP values and percentage mismatch which leads to the conclusion that they are most likely the same functions. So it is unclear how Julia, with a similar history, has resulted in a less accurate implementation.

Microsoft CRT library has two interesting results, `sinh` and `cosh`. `cosh` is correctly rounded whereas `sinh` is correctly rounded with the exception of one input value, 0.0005589425.

	Worst ULP			**Incorrect(%)**		
	Julia	MCRT	glibc 2.27	Julia	MCRT	glibc 2.27
sin	1	1	1	0.01	0.07	>0.00
cos	1	1	1	0.02	0.07	0.11
tan	1	1	2	7.10	0.01	1.95
exp	1	1	1	0.76	>0.00	0.01
expm1	1	3	1	0.53	6.95	0.53
exp2	1	2	1	>0.00	8.67	0.01
exp10	1	N/A	0	0.57	N/A	0.00
log	1	1	1	0.01	0.01	0.02
log2	1	2	1	0.54	0.69	0.01
log10	1	1	2	0.58	0.11	1.39
asin	1	1	1	0.20	0.12	0.22
acos	1	1	1	0.27	0.03	0.25
atan	1	1	1	0.15	>0.00	0.49
sinh	2	1*	2	3.29	>0.00	3.19
cosh	1	0	1	0.97	0.00	0.80
tanh	2	1	2	2.77	0.10	5.36
asinh	2	2	2	12.59	14.37	14.48
acosh	2	2	2	22.65	23.28	22.67
atanh	1	2	2	2.40	6.47	2.44

* MCRT `sinh` has only a single 1 ULP error mismatch and is otherwise correctly rounded.

Table 4.2: This table shows the Max-ULP accuracy, and percentage of mismatches with the correctly-rounded result for all valid input 32-bit floating-point numbers. The table shows `tan` having worst-case 1 ULP of error for Julia and MCRT but 2 ULP for glibc. However, the percentage mismatch shows even though MCRT and Julia have the same error in ULP they do not have the same percentage of correct answers, with Julia having approx. 7% of answers return with incorrect rounding compared to the 0.01% of MCRT.

The result is shown below:

$$\text{Correct Ans.} = 0\ \ 01110100\ \ 00100101000011000000000$$
$$\text{MCRT Ans.} = 0\ \ 01110100\ \ 00100101000010111111111$$

This is the result of the value being rounded down instead of up. This is one of the trickier numbers to find the correct answer for as a single change of the value of the last bit leads to the chain of flipping of ten other bits. This is almost half the 23 mantissa bits from a single rounding value change.

It was also unexpected to see up to 3 ULP in the Microsoft C Runtime as it is a library with ongoing development and that level of error could result in problems that are not as easy to be dismissed as 1 ULP.

The highest percentage of mismatches were found in the `asinh`, `acosh` and `atanh` functions where nearly a quarter of some functions return an incorrectly rounded result for their defined domain. The Julia implementation of `tan` also showed a high level of mismatches with 7% compared to the much lower values of other libraries. Figure 4.1 shows the distribution of the mismatches which were found in six functions. The distribution of the mismatches provides hints at the implementation. `sin` and `cos` for each library clearly appear to be shifts of each other and between libraries, we can see similar distributions which would result from similar implementations with some variation depending on the distance from $0.0f$. A good example of this would be the Julia and MCRT implementations of `log` which appears to have similar density of bands at the same positions — with the exception of Julia appearing to correctly handle inputs that return values near 0.0.

The combination of unique ULP, mismatch and distribution patterns poses an interesting security concern in the possibility of an information side-channel. With knowledge of the specific error distributions of each library, it would be possible for a user to fingerprint which maths library is being used by an application. This might reveal system properties to an attacker, for example, with the result from an MCRT function it may be possible for an attacker to identify if a remote machine running some software they can connect to is using that library and reasonably assume they are running a version of Microsoft Windows, which would help to identify which vulnerabilities may be open to that machine.

4.4 32-bit GPU Functions

While most libraries that are commonly used do not openly state how close they are to correct-rounding, NVidia does for their CUDA Toolkit. In their CUDA Toolkit v11.1.1 documentation, Section *F.1. Standard Functions* provides full tables of the mathematical functions with both single and double precision max ULP error.

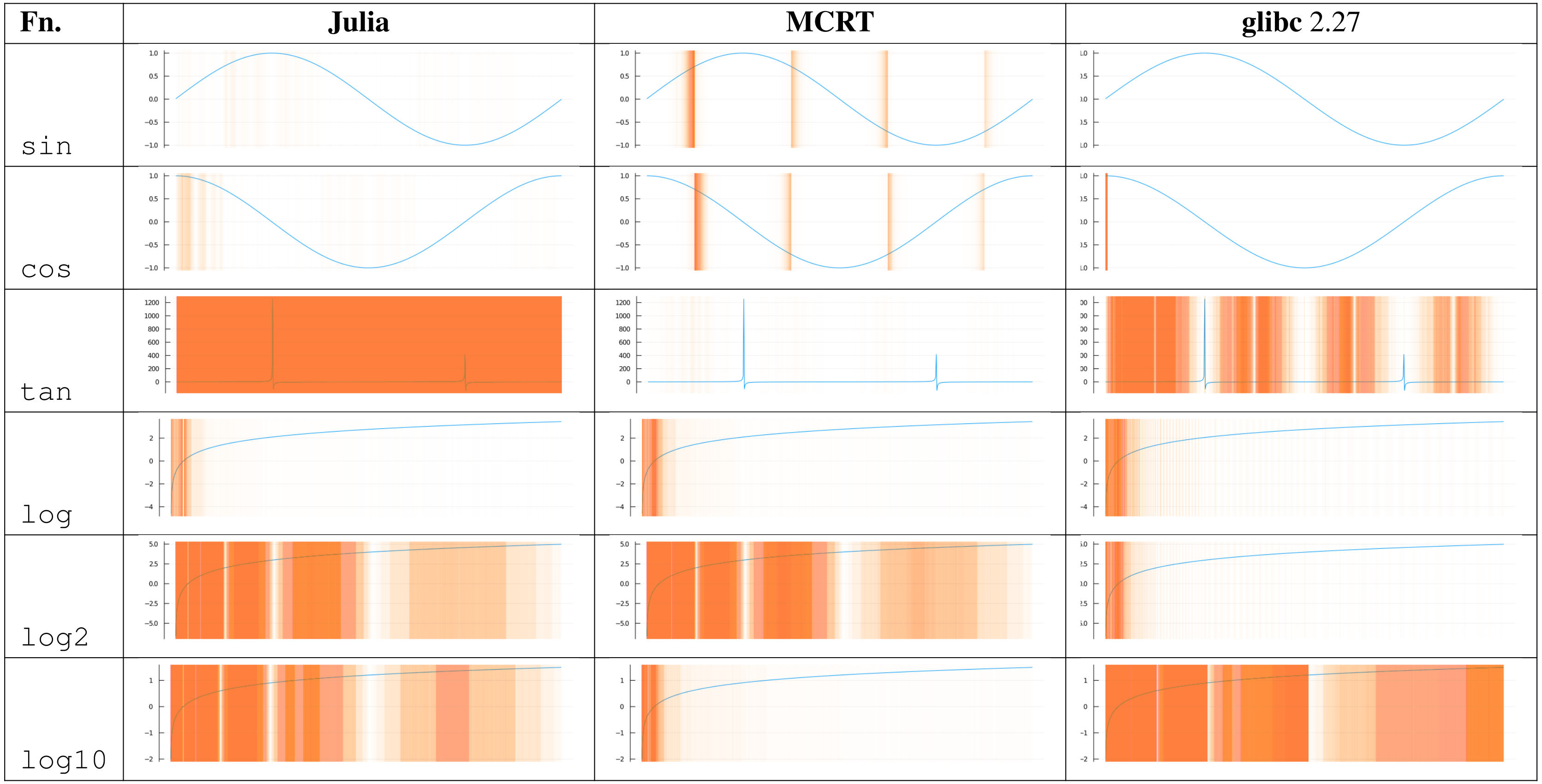


Figure 4.1: Sample of elementary functions showing the distribution of incorrectly rounded results on the CPU for Julia, MSVC and glibc 2.27. Different libraries produce error in different places, meaning $f(x)$ in one library might not be equal to $f(x)$ in another library for some inputs — this is a threat to portability. **NOTE:** *In this table a translucent vertical orange line is drawn for each incorrectly rounded result, for some graphs such as Julia* `sin`*, where there are relatively few incorrectly rounded results the errors are quite faint. This is needed so that most functions with lots of error are not all an orange bar.*

		NVidia Docs.	Our Results (NVidia 1080 GTX)		
Function	**Input Range**	**Max ULP Err**	**Max ULP Err**	**Mismatch Count**	**Incorrect (%)**
sinf	$(-\infty,\infty)$	2	1	426 015 575	9.96
cosf	$(-\infty,\infty)$	2	2	429 339 511	10.04
tanf	$(-\infty,\infty)$	4	3	836 917 130	19.56
asinf	$[-1,1]$	4	2	21 377 944	1.00
acosf	$[-1,1]$	3	2	162 888 434	7.64
atanf	$[-\infty,\infty]$	2	2	181 296 104	4.24
sinhf	$[-\infty,\infty]$	3	3	38 349 070	0.90
coshf	$[-\infty,\infty]$	2	2	65 355 938	1.53
tanhf	$[-\infty,\infty]$	2	2	39 501 004	0.92
log10f	$[0,\infty]$	2	2	763 162 985	35.68
log2f	$[0,\infty]$	1	1	12 044 078	0.56
logf	$[0,\infty]$	1	1	72 516 689	3.39
expm1f	$[0,32]$	1	1	16 148 303	1.46
exp10f	$[0,32]$	2	2	60 512 159	5.46
exp2f	$[0,32]$	2	1	60 454 446	5.46
expf	$[0,32]$	2	2	59 176 502	2.77

Table 4.3: Comparison between the reported max ULP error in the CUDA Toolkit documentation and our empirical results for all valid 32-bit floating-point numbers. The exponential functions are only tested for a subset of the IEEE-754 range due to out-of-range errors when testing larger inputs.

This data was validated for the functions which were analysed in the CPU mathematical libraries and can be seen in Table 4.3.

When analysing the results, the NVidia documented results appeared to be conservative and stated higher maximum ULP Error than was able to be reproduced (see `cosf`, `asinf` and `exp2`). Generally, it was found that the ULP error in the CUDA Toolkit mathematical functions had a higher max ULP error than the CPU equivalents and even when the max ULP is the same the percentage of incorrectly rounded results is significantly higher.

Looking at the error distributions in Figure 4.2, it is clear that the error is not only more frequent but also more consistent across the domain. Unlike the analysed CPU functions where the error diminishes as the input value increases. The error distribution in the GPU results would be a typical pattern of error for a function that has a less precise range-reduction algorithm in-place. However, as the CUDA Toolkit is closed source, it is only possible to speculate on the actual implementation.

These findings show that the problems mentioned about the portability of code will be more likely to appear when porting code to the GPU using CUDA due to the higher number of incorrectly rounded results which will not align with those of another library. This means that porting sensitive code from the CPU to the GPU using the CUDA framework is not trivial and

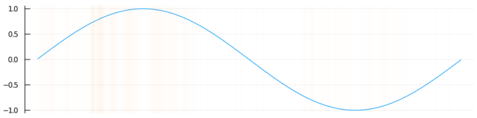
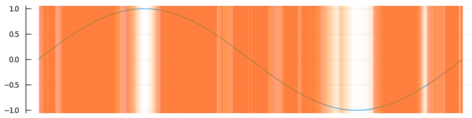
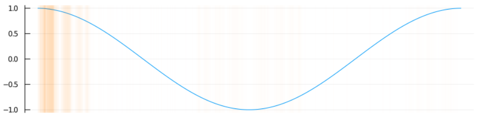
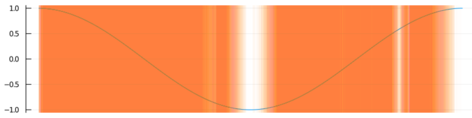
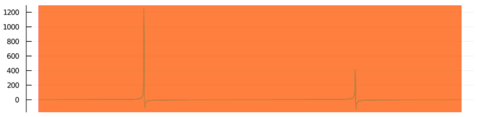
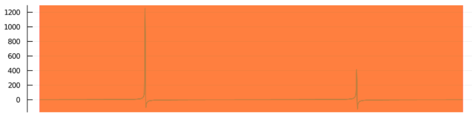
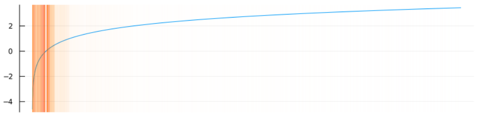
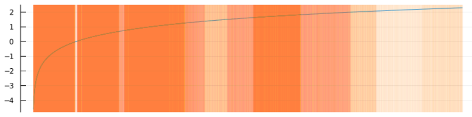
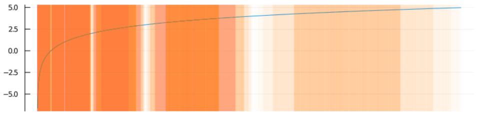
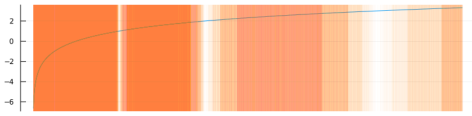
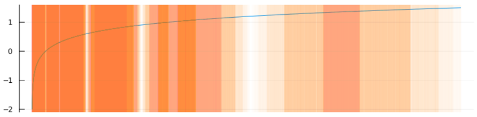
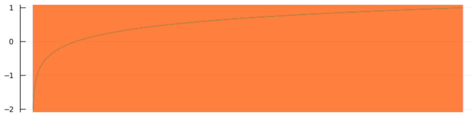
Fn.	Julia CPU	NVidia CUDAToolkit GPU
`sin`		
`cos`		
`tan`		
`log`		
`log2`		
`log10`		

Figure 4.2: Sample of incorrectly rounded results on the GPU using NVidia CUDA Toolkit compared to the Julia implementations on CPU.

there may be minor errors and inconsistencies related to the use of the supplied mathematical functions.

I believe this will be currently having an impact on machine learning frameworks that allow for CPU/GPU interoperability.

4.5 Monotonicity Testing

One concern when seeing high ULP error is that the error may result in incorrect-rounding in the wrong direction, causing a loss of monotonicity. This would cause problems with many hill climbing or gradient descent models which rely on the functions maintaining correct gradients.

All the specified 32-bit single-input functions which should be monotonic in Julia, MCRT and FDLibm were tested and it was found that they were all correctly monotonic in their domain (specified in Table 4.1).

4.5.1 Duplicated Results

Recommended functions are typically implemented with the input type being the same as the return type and most of the recommended mathematical functions are continuous. A result of this is that when the gradient of the continuous function is less than one there are more input values than output values in that range. This means that more than one input is mapped to one output. For example:

$$\sin(1.5939530) = 0.9997319$$
$$\sin(1.5939529) = 0.9997319$$
$$\sin(1.5939528) = 0.9997319$$

When closely observed it has frequent plateaus of two or three repeated values, this is especially evident when the input is below one where the ratio of input to output values is $1:1$ so any gradient below 1.0 is guaranteed to produce repeated results. In the $[0.0, 2.0\pi]$ range of `sin` 0.7% of all results are part of a duplicate (see Figure 4.3).

As the input values increase there are fewer floating-point values in the range of inputs and therefore the number of duplicates is reduced until it is only present where the gradient is equal, or is very close, to zero.

Figure 4.3 shows the areas of duplicated results and how they quickly diminish as the ratio of input to output changes.r

These duplicated value plateaus can affect programs that rely on gradient descent where it may possibly not correctly calculate the gradient when given only a small window for differentiation.

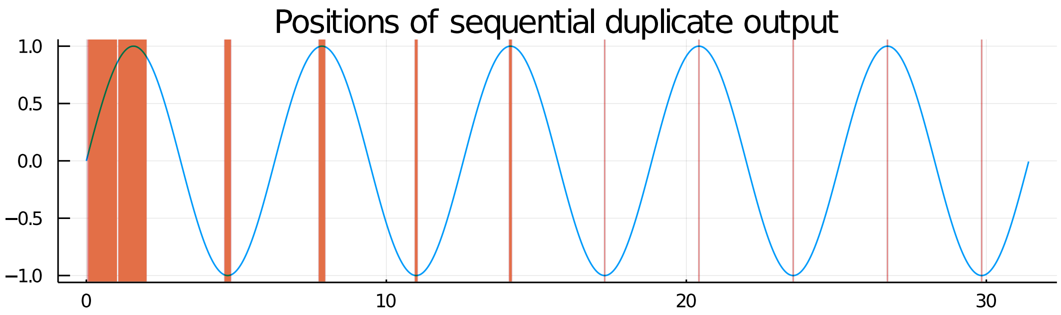


Figure 4.3: Locations of sequential equal output values. Showing how the number of duplicates decreases as the ratio of input to output values for each period of the 32-bit `sin` function.

Unlike other problems addressed in this chapter, this one can be shown to also affect higher precision types. Floating-point numbers are affected in this situation because there is more precision required in the output than the input when the gradient is less than one and the number of representative values for input is equal or larger than the output. As a result, this is a problem that cannot be solved by moving to a higher-precision representation.

One solution that is available to address this limitation in the representation of the recommended functions is to allow for higher-precision outputs from lower-precision inputs when the function is within known areas of misrepresentation by duplication. This is acceptable under the current IEEE-754 standard as the input and output types are not restrictively defined. With this approach, a 32-bit function would return 64-bit but only compute greater than 32-bit when in an area of known duplication. This would provide the same performance for the vast majority of inputs but allow for the correct behaviour of algorithms that rely on accurate gradient calculation for a particular function. To ensure this the areas of duplication were tested in 64-bit for all 32-bit floating-point inputs in that area and it was found to produce zero duplication, meaning there is a continuous gradient of at least 32-bit precision.

4.6 Examples of Error

This section covers the real-world examples of where being correctly rounded does result in meaningful change in the output value. There is a focus on visual results to be able to more easily convey the divergence.

4.6.1 Mismatch Example

The first example of this section of real-world problems caused by not following the standard is loss of inter-library determinism. Table 4.4 shows the function `tanh` implemented in three different libraries and each result is different. If these were three different systems trying to find the same answer using different libraries - they could not have a consensus. In this example, Microsoft CRT provides the correctly rounded result and the two other libraries (Julia and glibc 2.27) round in opposite directions.

With this error in place, safe distributed computing across libraries is not possible for any calculations which need bit perfect results.

Function	**Input (32-bit float)**	**Result (binary32 bit representation)**
tanh (CRLibm)	0.058622412	0 01111010 11011111110101110111100
tanh (Julia)	0.058622412	0 01111010 11011111110101110111011
tanh (MCRT)	0.058622412	0 01111010 11011111110101110111100
tanh (glibc 2.27)	0.058622412	0 01111010 11011111110101110111101

Table 4.4: This table shows a single input to the `tanh` function giving different results from our main testing libraries. Microsoft CRT is correctly-rounded where as Julia and glibc both round in opposite directions.

4.6.2 Procedural Example

Procedural generation techniques are used to generate data programmatically. They are commonly used in computer graphics for everything from generating textures to whole environments. Outside of this, they are also useful in computer vision, path-finding and other research areas which rely on generating a large amount of realistic content for testing. Reproducibility is a major factor in procedural generation. Generated new data must be able to be recreated from the same inputs again regardless of the device on which it is being computed. Without this guarantee, it is impossible to reliably use the procedural content as the output may not be what was expected.

The video game Elite:Dangerous is a known example of widespread and complex procedural content being used across different platforms. It is set in a virtual procedurally generated 3D scale model of our galaxy [21]. Due to the reproducibility nature of the procedural generation, many players can play the game in the same 'world' without having to transmit the entire map to each other or worry about getting mismatching results. This gives the users a larger play space than would be possible without procedural generation.

That is possible because the procedural techniques, in this case, repeated combinations of different procedural noise patterns being used, such as *Perlin Noise* [126], are stable, i.e. they can be implemented to produce the same results in different environments. If they were not stable, as is the case for many elementary functions, then the combination of multiple levels of procedural noise pattern (such as Perlin Noise) would result in increasingly divergent results between machines.

The field of procedural generation has many different approaches for mathematically reproducible output – but for commercial applications, some of these may result in an error when being used due to the differences in output from the elementary functions and it is something programmers must consider.

Consider the *Chaos Game* [10, 87], a method of creating fractal patterns used in the study of fractals and strange attractors [114, 7, 11, 45]. A pattern is generated by taking an initial position and then applying a function to it to produce a new position. Over many iterations, this results in a pattern. These types of iterative procedures are common in procedural generation in computer graphics [145]. If the function applied in each step produces different output on different systems then the final output is not equal.

$$p_{n+1} = f(p_n)$$

It was found that when using elementary functions which do not conform to the IEEE standard the final output images of the *Chaos Game* converged on different outputs to the correctly rounded solution. This effectively makes these approaches non-portable.

Figure 4.4(a) shows the result of a *Strange Attractor* which uses both `sin` and `cos` in it's calculations.

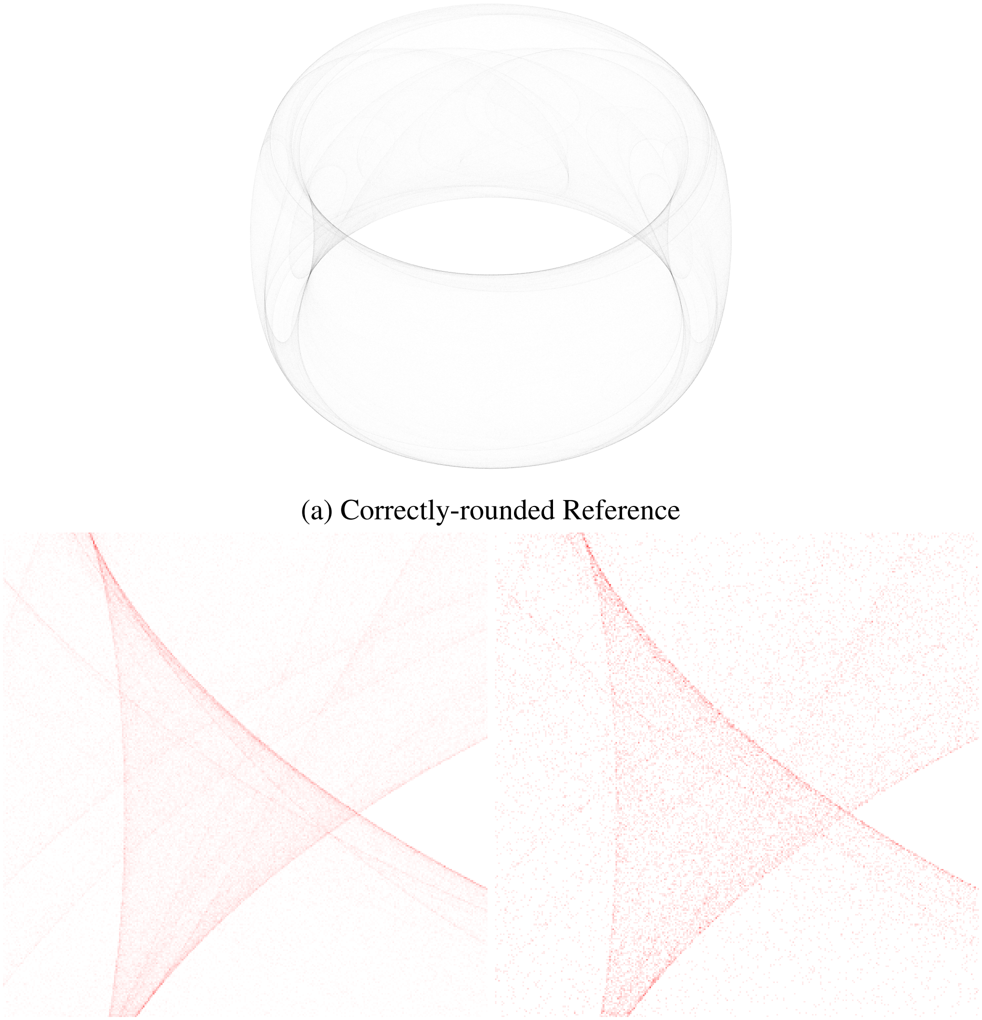
$$\begin{aligned} x_{n+1} &= d * sin(x_n * a) - sin(y_n * b) \\ y_{n+1} &= c * cos(x_n * a) - cos(y_n * b) \end{aligned}$$

The incorrectly rounded solution is noisier than the correctly rounded solution. It has failed to converge as evenly and appears to show clustering patterns (Figure 4.4) inconsistent with the correct result. The erroneous result is caused by repeated missteps due to the small errors in the `sin` and `cos` calculations.

The *Reaction-Diffusion* algorithm is commonly used to create organic-looking patterns, wave simulations and procedural noise. This algorithm uses a convolutional kernel with two buffers to 'grow' a pattern from an initial state. It can be used to create organic-looking patterns, wave simulations and procedural noise.

Figure 4.5 shows that when the underlying elementary functions do not match it leads to compounding errors that cause divergence in the pattern being generated. Unlike the Chaos Game example where the error simply resulted in a lower quality image, the error here results in a different result between implementations.

By not being correctly-rounded the uses of procedural generation may have been limited, as only approaches which do not use the elementary functions, such as *Perlin Noise*, can be used in real applications that require reproducibility or portability between systems.



(a) Correctly-rounded Reference

(b) Correctly-rounded Sample

(c) Julia Sample

Figure 4.4: This shows a zoomed in view of the edge of part of the strange attractor in the Chaos Game demonstrating how a small amount of error in the functions can lead to a failure to correctly converge and form the smooth distribution. It shows clustering in the Julia implementation that occurs due to the incorrect values of the Julia implementation.

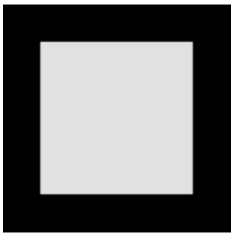
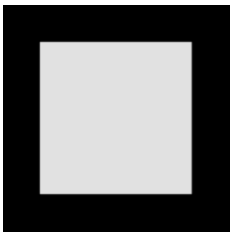
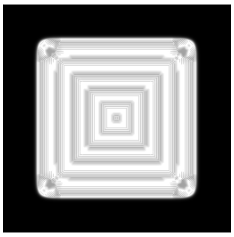
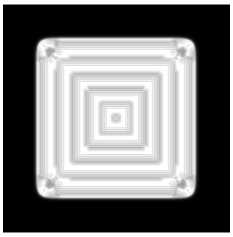
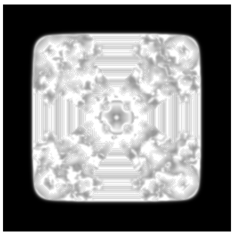
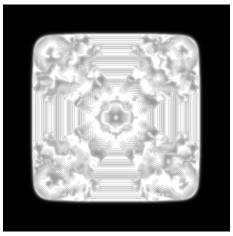
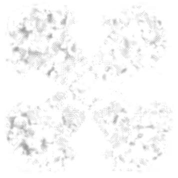
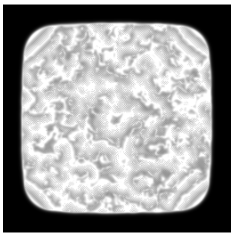
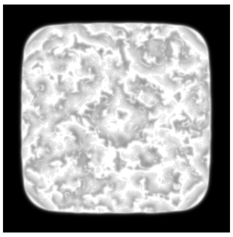
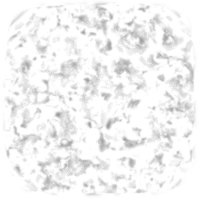
Step	**Julia**	**MCRT**	**Difference**
1			
100			
200			
500			

Figure 4.5: This table shows the Reaction-Diffusion Algorithm using `tanh` to generate wave patterns with the Julia and Microsoft CRT libraries. It demonstrates the lack of portability of the code as it results in different outcomes from different libraries. The simulation diverges around step 100 and begins to produce visually dissimilar outputs. If the IEEE standard is followed, this simulation would not diverge.

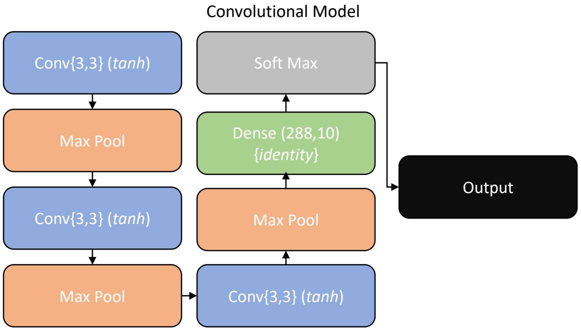


Figure 4.6: The convolution model design that was used to produce differing results when the 'tanh' function is implemented with different libraries.

4.6.3 Neural network example

Key to algorithms used in neural networks is a function to provide some non-linearity, these are called activation functions [123]. For this example we are concerned with the `tanh` elementary function, a standard function recommended for implementation by IEEE-754. This activation function is integral to the popular LSTM (long short-term memory) [62] and convolutional networks and is widely used [74, 152, 75]. Activation functions are used in the training of models as well during inference. Models in machine learning often use lower-precision floating-point numbers, such as 32-bit floating-point, for performance reasons [79] and for GPGPU support [5].

It is common for the weights of a model to be trained on one machine and then transferred to another for either further training or to be used for inference.

It is possible to show that training using `tanh` from different implementations results in different weights in the resulting network. This was tested using a convolutional neural network where the activation on the convolutional layers is `tanh`. The shape of the network that was used can be seen in Figure 4.6.

It was also found that placing the weights from a model trained with implementation A into a model using implementation B may result in differing one-hot encoded results.

4.6.3.1 MNIST Example

MNIST is a standard dataset for training the recognition of hand-written numerical characters. Figure 4.6 shows a commonly-used design of model for classifying the MNIST dataset. I created two versions of this model and both are seeded with the same starting weights. One uses the Julia implementation of `tanh` as the activation function in the convolutional layers, while the other uses the Microsoft CRT implementation.

They are both initialised with the same starting weights and optimiser state. They are then trained for one epoch with the MNIST dataset [39] and then the outputs are compared and the number of misclassifications are then counted.

The two implementations resulted in different overall accuracy. This shows that the produced model is meaningfully different despite being composed of what should be interchangeable parts. The two versions were also not interchangeable, weights for each model are then swapped and the model is tested again. This results in different values for the resulting one-hot encoded outputs as shown in Table 4.5.

4.6.3.2 LSTM Example

LSTM networks are commonly used when the next inference result will be dependent on past results, or other scenarios where some sort of memory-like behaviour is desired in the model. These types of models are particularly susceptible to diverging behaviour when the activation functions being used is changed from one implementation to another. Any difference in result are now stored and can potentially impact each subsequent result in a chain, leading to greater divergence over time.

To test this, models were implemented with three LSTM layers and a final densely connected layer. Each implementation had different sized LSTM layers (32, 64, 128 and 256). Each model was trained to reproduce text similar the sample text it was trained on. Once the model was trained and outputting fairly coherent text the model was then used to generate a large sample text. The generation of the sample text was then repeated with the `tanh` activation function used in training changed to the `tanh` function from another library. The differences in the generated text were then compared.

It was found that after switching the activation function the early results remained the same, but the longer that the LSTM was run the more frequently results would be different, resulting in higher occurrences of differing output characters. The rate of this divergence and whether the error resulted in a different output varied depending on how long the model was trained. With a relatively untrained model the weights are quite random, meaning the difference between picking one output character and another is, on average, high relative to the error in the LSTM occurring from small differences in the activation function output. However, it was found that the occurrences of different character output increased as the network was further trained and the difference in weights used for the selection of one output or another would be small.

This hints at the possibility that for LSTM networks based on text generation, or similar tasks where there can be multiple valid outputs for a given scenario, may be more likely to exhibit non-deterministic results caused by the unexpected differences caused by the different activation functions used.

More generally, this shows that LSTM networks are non-deterministic when they are linked to mathematical functions from different libraries.

MNIST ID	$\text{tanh}_{\text{julia}}$	tanh_{cr}	Diff.
0	`-0x1.78f706p-1`	`-0x1.78f6fap-1`	0x1.8p-22
1	`0x1.5e8a0cp+2`	`0x1.5e8a12p+2`	0x1.8p-20
2	`0x1.a710acp+3`	`0x1.a710a4p+3`	0x1p-18
3	`0x1.1ceff6p+2`	`0x1.1ceffcp+2`	0x1.8p-20
4	`-0x1.f960aap+2`	`-0x1.f960a2p+2`	0x1p-19
5	`-0x1.821d5ep+1`	`-0x1.821d54p+1`	0x1.4p-20
6	`-0x1.f2787p+2`	`-0x1.f2786cp+2`	0x1p-20
7	`-0x1.418a82p-2`	`-0x1.418aa2p-2`	0x1p-21
8	`-0x1.fbab0ap-1`	`-0x1.fbab0ap-1`	0x0p+0
9	`-0x1.9176d8p+1`	`-0x1.9176d6p+1`	0x1p-22

Table 4.5: MNIST Classification Example. This shows the difference in a one-hot encoding of a convolutional neural network trained using Julia's tanh function and then evaluated with that function and a correctly rounded implementation.

4.6.3.3 Neural Network Conclusion

The different result and changing behaviour when weights are transferred is a worrying sign for quiet errors which may exist in current system which rely on weight portability or continuous training across different devices. As it is fairly common for models to be deployed across different platforms (particularly in heterogeneous mobile devices), it could result in some deployed machine learning systems working with small errors that may be unknown to the developers as this is assumed to be correct.

This is particularly a concern as machine learning starts to be used in safety critical applications such as self-driving cars and medical diagnostic tools. It is essential that such use-cases can be rigorously tested and that may not be possible if they are relying on disparate implementations which, unknown to the developers, do not produce the same results.

4.7 Non-Determinism Between Library Versions

Non-determinism is a problem beyond inter-library operability. It can also be a problem between library versions. In this section, it is showm how a change to the range-reduction algorithm used for the 32-bit `tan` function in glibc between versions 2.27 and 2.31 causes different results when running Ubuntu 18.04 and 20.04.

In the upgrade from Ubuntu 18.04 to 20.04 the version of glibc was changed and this means to a user who upgrades their system they will be getting a change in the functionality of a key mathematical function invisibly, and likely, unexpectedly.

4.7.1 The Change

The change to the range reduction function in glibc was made in August 2018 [41]. The widely used function `__ieee754_rem_pio2f` was replaced with a version named `rem_pio2f` [42] stored locally to the `tanf` function. Instead of the very large and complex behaviour of `__ieee754_rem_pio2f` [43], the new implementation relies on the accuracy of double-precision to calculate a result with correct-rounding during the reduction when the input is below $120.0f$, otherwise, it uses a more expensive method similar, but still simpler, than the old function.

By changing the resulting value of some range-reduced values, the tangent function kernel which takes the reduced values then produces different (in this case, more correct) results.

4.7.1.1 Analysis of Change

In Figure 4.7 each incorrectly rounded input in the $[0.0, 2.5\pi]$ domain for each library version is rendered as an orange vertical line. This shows the distributions of incorrectly rounded results.

By comparing the two library versions, the difference becomes apparent around the $\frac{\pi}{2}$ intervals. This means that when calling the same code linked against glibc 2.27 instead of 2.31 then some inputs for `tanf` near the $\frac{\pi}{2}$ interval may return different values — meaning the result of the function can not be determined when compiling the same code on machines using different versions of the glibc mathematical library. Effectively making generic use of the mathematical functions non-portable.

4.7.1.2 ULP Error of `tanf`

Table 4.6 shows that after the 2.27 release the worst case ULP error has been reduced (in round-to-nearest mode) to one ULP error in versions 2.31 and 2.32, down from two in 2.27.

This is the result of some of the worst approximations being improved for inputs values close to the $\frac{\pi}{2}$ interval and a number of previously incorrectly rounded results are now correctly rounded.

4.7.1.3 Differences Between Versions

By testing all positive floats in `tanf`'s domain, Table 4.6 shows that between the 2.27 release and 2.31 release there is a difference of approximately sixty-nine-thousand fewer incorrectly rounded results. Figure 4.7 shows where error is distributed for the function in general, and where these sixty-nine-thousand outputs have been corrected.

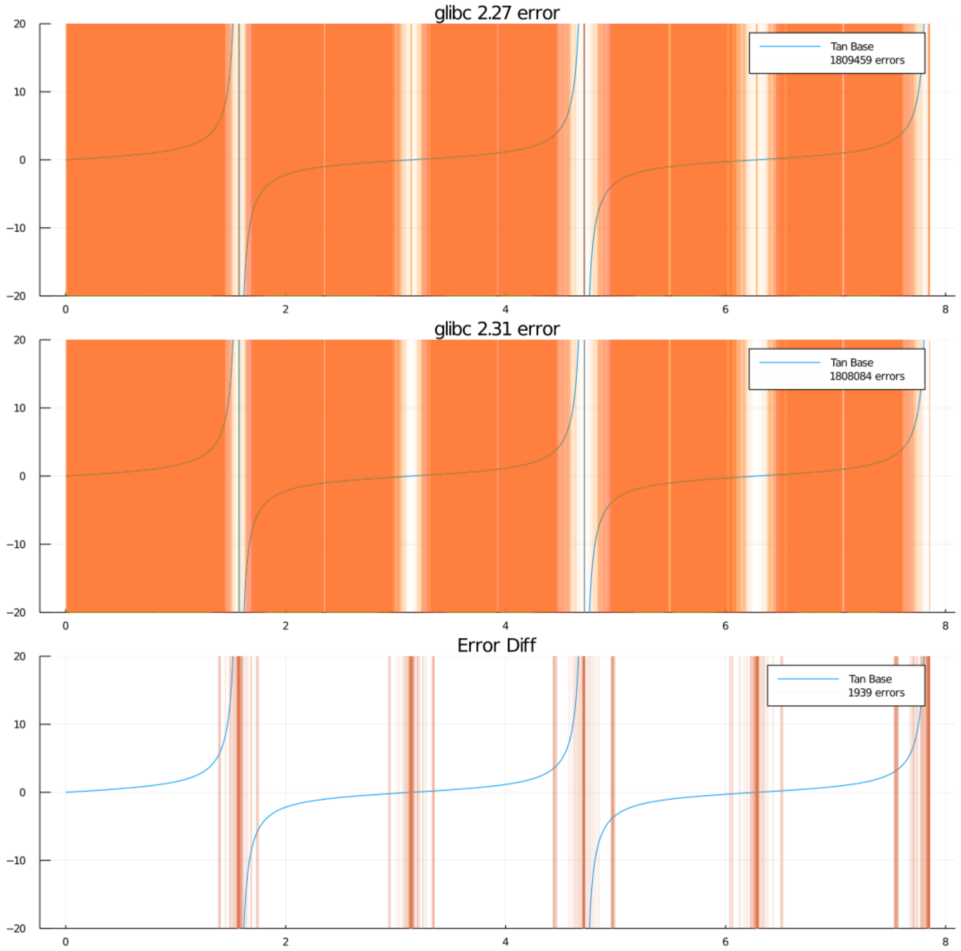
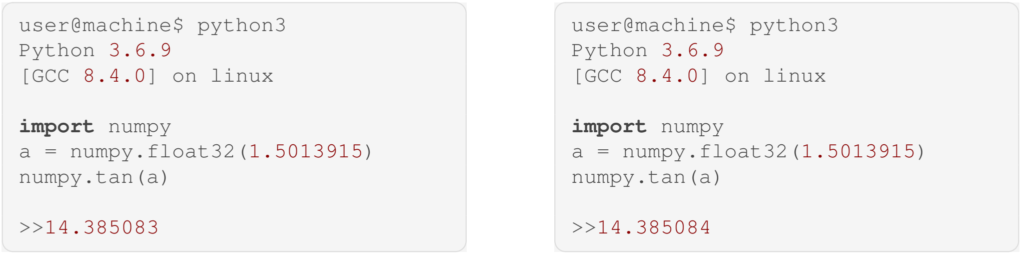


Figure 4.7: Here it can be seen that the change to the `tanf` implementation has resulted in slightly fewer errors in the 0.0 to 2.5π range. The difference is shown in the lower graph where most error was removed at multiples of $\frac{\pi}{2}$. These are the reflection edges of the piece-wise `tanf` kernel.

Version	Max ULP Error	Mismatch Count	Difference
2.27	2.0837	41,774,574	-
2.31	1.4755	41,705,625	68,949
2.32	1.4755	41,705,625	0

Table 4.6: Testing for the ULP error and counting incorrectly rounded results for all inputs in `tanf` the positive domain.



```
user@machine$ python3
Python 3.6.9
[GCC 8.4.0] on linux

import numpy
a = numpy.float32(1.5013915)
numpy.tan(a)

>>14.385083
```

(a) Running with gnulibc 2.27

```
user@machine$ python3
Python 3.6.9
[GCC 8.4.0] on linux

import numpy
a = numpy.float32(1.5013915)
numpy.tan(a)

>>14.385084
```

(b) Running with gnulibc 2.31

Figure 4.8: This example shows running Numpy and using one of the mismatch values identified in gnulibc to produce different results on Ubuntu 18.04 and Ubuntu 20.04 for the same input.

4.7.1.4 Real-World Example

This has a real impact if two users are running different versions of an operating system and expect the same result. Take for example SciPy, the framework for scientific computing. It is important that any two users produce the same answers when using this application because it is used in areas where accuracy is critical to reproducibility. SciPy wraps Numpy, a library for matrix and other high-level mathematics in Python, and Numpy doesn't provide its own mathematical library implementation. Instead, it relies on dynamically linking to the mathematical library on the system where it is installed.

Figure 4.8 shows how this can go wrong using one of the `tanf` mismatches between gnulibc 2.27 and 2.31.

4.7.1.5 glibc `tanf` Summary

The change to `tanf` was able to significantly improve the worst-case ULP accuracy of the function and removed some incorrectly rounded results from the algorithm. The commit message also reports speedups due to better code inlining with the new implementation. These are all important (and impressively tidy) changes that are desirable in a mathematical library.

However, due to the library not following the correct-rounding aspects of the IEEE-754 standard, this change has resulted in a breaking difference between the two versions which is likely unknown, undocumented and highly unlikely to be checked by the end-user.

It has made it so that `tanf` in 2.27 is both intensionally and extensionally different to the `tanf` implemented in 2.31. As a result, a user upgrading their Ubuntu operating system from 18.04 to 20.04 and then recompiling some code they were working on which uses `tanf` would result in a new executable which produces different result in subtle and hard to identify ways.

This is likely not a desirable trait for a mathematical library that is so widely shipped and used. The error caused by this change is likely small and would be expected to not affect the final output of many applications but it is never the less, a risk to anyone who uses it.

```
float f()
{
    return std::tan(1.5013915f);
}
```

(a) C++ input

```
f():
  movss xmm0, DWORD PTR .LC0[rip]
  ret

.LC0:
  .long 1097214286
```

(b) Compiled Output

Figure 4.9: Example of the *Constant Evaluation* optimisation of mathematical functions in gcc 10.2

4.8 ULP Error, `constexpr` and Non-Determinism

Currently, proposal P1383R0 [135] is being considered by the ISO C++ Standards Committee. This document proposes that the functions in the `<cmath>` and `<complex>` headers should be made to have the C++ attribute `constexpr`. This attribute makes the compiler evaluate the function at compile-time and only store the result of that computation in the output instructions. As seen earlier in this chapter, most mathematical libraries are not producing correctly rounded results at run-time. If the libraries cannot agree on canonical results for an input, static evaluation of that mathematical function will result in mismatches between the static and run-time result which may cause unexpected behaviour.

This section shows why using the `constexpr` attribute with existing mathematical functions implementations would be likely to cause problems. It has been showm that the existing *Constant Folding* optimisation is already partly implementing this feature despite it being against the rules defined in the C++ Standard. It is arguable that this is an existing case of mathematical error/approximation that is being tolerated within all mathematical programs which do not statically link to the library which is used at compile-time to generate the static values.

The work showing that the current widely-used mathematical libraries being incorrectly rounded makes me concerned about the committee proceeding with this proposal before that problem is solved because it would mean a programmer could use `constexpr` to compile the result of a mathematical function into a library at compile-time and then use the result of that calculation at run-time alongside values computed at run-time potentially with a different library. If the device which performed the compilation is linking to a different mathematical library than the library which is being used at run-time, the one or more ULP error in some mathematical functions could result in the strange situation where $\sin(x) == \sin(x)$ will return false! The program result will be non-deterministic as a result of the environment where it is ran — which is one of the problems which the IEEE-754 expressly aimed to avoid in 1985.

This is only a proposal and concerns have been raised with the committee, however, if you compile a program now in `gcc` you will see that mathematical functions are already being optimised to constants during the *Constant Evaluation* optimisation pass, shown in Figure 4.9.

Compiler	**Versions**
X86/X64 msvc	None
Clang	3.8 -> Current
*gcc**	4.4.7 -> Current
Djgpp	4.9.4 -> Current
KVX	7.5
Raspbian Buster	Current
Raspbian Stretch	Current
FRC 2019	Current
FRC 2020	Current

Table 4.7: List of compilers and the first version found with *Constant Evaluation* of the standard mathematical functions. *Compiler Explorer* [66] was used to quickly evaluate a large number of compilers and compiler versions but the list is not exhaustive for all released versions.
(*) In gcc constant folding with MPFR was introduced in 4.3.0, but this version of gcc was not available in the test suite used.

This behaviour was unexpected as the mathematical functions as defined in IEEE-754 raise exceptions — which are a side-effect. Side-effects are not allowed to be disregarded in *Constant Evaluation*[2] as this would change the behaviour of the program. They can be disregarded if they are proved to be not observed. This was tested in gcc 10.2 and, even when trying to catch exceptions around the function call, the compiler still reduced the function call to a constant.

Many C++ compilers have been tested to find which ones exhibit this troublesome optimisation, shown in Table 4.7. It shows that the vast majority of popular compilers are performing this action. The exception being much older versions, and X86/X64 Microsoft Visual C++ compiler.

On the surface, this is just a standard non-compliance issue which is fairly common and compilers have many. What makes this interesting is that compilers rarely implement their own mathematical functions. This means that if the compiler is dynamically linking to the library for mathematical results the compiler is unaware of the specific implementation of the function it is calling to produce the replacement constant and therefore has no guarantees that the function is side-effect free.

This produces two problems, firstly it means that if a library being linked is not correctly rounded and then the produced executable is dynamically linked against another library, or library version, there is a chance that $\tan(x) \neq \tan(x)$ when one is evaluated at compile time.

Secondly, it is a security issue. It allows someone to change the results being sent from an external library into the compiler. Which not only compromises the compiler but any programs it might generate. This is not a new problem, but the attack surface is further increased when this behaviour will be moved from a compiler specific optimisation that is not in the standard to something the C++ standard will require in scenarios where `constexpr` is used.

[2] Sometimes called *Constant Folding*.

Demonstrating the first problem is quite trivial. You can write a program to call a mathematical function with a known value that is different in two libraries and disable optimisations on one of the calls. This will prevent your compiler from evaluating both at compile time. Add in a comparison that the two functions are equal and then compile the program. On the machine the program was compiled on, using the same linked library that was used by the compiler to generate compile-time result, the functions should report as equal. If you move the executable to a machine with a different mathematical library (or change the link location on the current machine to point at a different library) and rerun the program you may see the program report that $\tan(x) \neq \tan(x)$.

Figure 4.10 demonstrates this using an input to `tanf` which is known produces different results in gnulibc 2.27 and gnulibc 2.31 so that the disparity of output between Ubuntu 18.04 and Ubuntu 20.04 can be tested.

Another interesting quirk is that Clang has a bug in its constant folding code (as of version 11). If you compile and run the shown test code with Clang on Ubuntu 18.04 you will get a mismatch, even though you both linked to the same library.

The bug is caused by Clang compile-time evaluating `tanf` as a double-precision floating-point function and then rounding the result back to 32-bit. This causes Clang to emit the correctly rounded result whereas at run-time glibc 2.27 on Ubuntu 18.04 will produce an incorrectly rounded value, resulting in a mismatch.

As the correctly rounded result cannot be guaranteed at run-time with the current libraries available, and some compilers are outputting incorrectly rounded results, it seems that *Constant Evaluation* optimisations and `constexpr` for mathematical functions should be placed behind the existing `-ffast-math` flag which is there to allow for optimisations which are non-compliant with ISO C/C++ and IEEE-754. Without this change users are relying on the programs which take advantage of this optimisation being error-tolerant up to the value of the difference between the static compilation mathematical library and whichever library is linked at run-time. Most users will not be aware that there is a chance of error here as it is not signalled or documented and it is not possible to know what will be linked at run-time as there is a variance in compliance with the standards which dictate the behaviour of the mathematical libraries. All together, that makes this an unsafe mathematical optimisation for the majority of the libraries which are currently commonly being used.

4.9 Conclusion

The 32-bit, single input, recommended functions from IEEE-754 feature in many applications on CPUs and GPUs. Four popular libraries were investigated and it was demonstrated that none of them deliver correct rounding in all cases.

```cpp
#include <cmath>
#include <iostream>

float f()
{
    return std::tan(1.5013915f);
}

// Prevent Constant Folding / Constant Evaluation
float g() __attribute__ ((optnone));
float g()
{
    return  std::tan(1.5013915f);
}

int main()
{
    std::cout << "constexpr: f() = "
              << std::hexfloat
              << f()
              << std::endl;
    std::cout << "runtime:   g() = "
              << std::hexfloat
              << g()
              << std::endl;
    std::cout << "f() == g()    := "
              << ( (f()==g()) ? "TRUE" : "FALSE")
              << std::endl;
    return 0;
}
```

(a) C++ Test Code

```
constexpr: f() = 0x1.cc529cp+3
runtime:   g() = 0x1.cc529cp+3
f() == g() := TRUE
```

(b) Output when compiled against gnulibc 2.31 and runtime linked to gnulibc 2.31

```
constexpr: f() = 0x1.cc529cp+3
runtime:   g() = 0x1.cc529ap+3
f() == g() := FALSE
```

(c) Output when compiled against gnulibc 2.31 and runtime linked to gnulibc 2.27

Figure 4.10: Example of *Constant Evaluation* breaking because of incorrectly rounded results in common mathematical libraries.

It seems that FDLIBM's (pre-standardisation) approach of 1 ULP worst-case error is almost ubiquitous. One way to interpret these results is as an operational definition of the required accuracy, in the next chapter is it shown how one can exploit this to improve performance within the same error bound. However, some concrete examples of this were given where this will have an impact on the application and so in Chapter 5 I show how correctly rounded implementations can be delivered with comparable performance.

Chapter 5

Correctly Rounded 32-Bit Elementary Functions

In the last chapter we showed that for some applications a correctly rounded implementation is necessary to produce satisfactory results and the currently provided mathematical libraries are non-compliant with this requirement. This chapter aims to fill this requirement by providing an implementation for correctly rounded 32-bit results for the mathematical functions.

Through some early testing we found that for many 32-bit functions the correctly rounded result can be found by simply casting the result from the 64-bit implementation to 32-bit. There are a small number of inputs where this does not produce the correctly rounded result and these can be stored and used as special cases. This results in an implementation of 32-bit correctly rounded mathematical functions which are not as expensive as some of the more numerically complex correctly rounded implementations and with similar performance to existing 32-bit implementations on modern CPUs which provide both 32- and 64-bit floating-point hardware evaluation.

The motivation for this work is that the current 32-bit implementations of common mathematical functions are not correctly rounded which results in them not being inter-operable and unreliable when the precise answer is expected. A justification for the current implementations being approximations was given as performance. This means that there exists at least two use-cases, those which require correctly rounded results and those which do not. The current correctly rounded implementations which are available [35] are many orders of magnitude slower than the 32-bit approximate implementations and in some cases do not even support direct 32-bit output (such as CRLibm).

The approach taken here is to analyse the 64-bit results of a specific implementation when they are cast to 32-bit to provide a list of the known input values which produce an incorrectly rounded result. For these specific inputs, as they are few, it is possible to simply perform a table lookup to return the correct result. If the input is one which does correctly round down from 64-bit floating-point then we cast to 32-bit float and continue.

This implementation is tested for all 32-bit inputs and performance results are compared to existing 32-bit functions to ensure that the implementations are practical and sensible for a modern application.

5.1 Current Approaches

When wishing to achieve high-precision correctly rounded results for standard floating-point mathematical functions there are only a few major libraries that are popular and currently being used in commercial applications. They are CR-LIBM (Correctly rounded Libc) [35], GMP (GNU Multiple Precision Arithmetic Library) [72], the GMP derived projects MPFR (GNU Multiple Precision Floating-Point Reliable Library) [53] and MPIR (Multiple Precision Integers and Rationals) [64].

This work focuses on CR-LIBM and GNU MPFR as they are shipped with Julia.

5.1.1 CR-LIBM

CR-LIBM is designed for provable and well performing implementations of 64-bit floating-point mathematical functions.

The library is based on the work by Lefèvre and Muller who proposed solutions for the maximum precision needed to compute the correct-rounded result for several functions [101] — although only in 64-bit precision.[1]

The library's approach is to prove the correct results for a function by calculating a tight bound on the total relative error. While they provide many proven results, some functions are only correctly rounded within certain bounds.

With this method, they can provide a bound on the 'Table-makers Dilemma' for many functions in 64-bit precision.

5.1.2 GNU MPFR

An alternative to the CR-LIBM approach is to use arbitrarily high-precision to establish a finite bound of correct-rounding by moving this bound further and further out — what CR-LIBM refers to as the "bounds of astronomy" [35].

With this method, the probability of producing the correctly rounded result is increased as the precision is increased. With the information from CR-LIBM this gives some guarantees of correct-rounding for some functions but others are only given a high probability of correct-rounding.

For those where correct-rounding can be achieved the "onion peeling strategy" (also known as Ziv's strategy) is used. This approach uses increasingly higher-precision in the approximations

[1] These were: the exponential, logarithm, sine, cosine, tangent and arctangent.

of the target function until the error bounds are such that the correct-rounding direction can be definitively chosen [173].

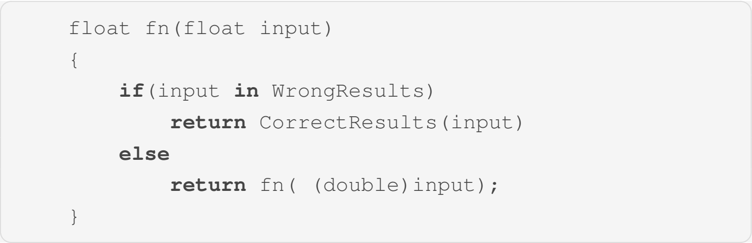
GNU MPFR has been used as the high/multiple-precision library for Java to produce high-precision and correctly rounded results [20].

Unlike CR-LIBM, GNU MPFR is multiple-precision and therefore supports 32-bit direct output. However, due to its programmable nature it is less performant than CR-LIBM and the standard library implementations.

5.2 The Approach

My approach to solving the correct-rounding problem for 32-bit is to run the 64-bit implementation of a function for all 32-bit inputs, round the number to 32-bit precision and then compare to the correctly rounded results from a correctly rounded implementation. If the results do not match, the input and correctly rounded output are saved into a table.

Once the full range of inputs has been tested, the table is used to create implementations in the form:



```
float fn(float input)
{
    if(input in WrongResults)
        return CorrectResults(input)
    else
        return fn( (double)input);
}
```

This means that getting the correctly rounded 32-bit result will be the cost of checking if the current input is a known incorrect 64-bit implementation value and then either the cost of calling the 64-bit implementation or the cost of fetching the correct return value.

In an ideal situation, as the number of incorrectly rounded results is low, the 64-bit implementation of the function should be on the fast path and the branch predictor will reduce the cost of the input value checking.

The above example shows a lookup into a *WrongResults* tables, for functions with a table of a small number of entries this can be replaced with an if-else statement for each condition to further help the compiler to optimise the code.

5.2.1 Debunking Simple Approaches

I approached the creation of this library by noting that for the $[0.0, 2.0\pi]$ range it is possible to achieve correctly rounded 32-bit `sin` by simply casting the result of 64-bit `sin` to 32-bit. In this range, the difference between the two results is not significant enough to result in incorrect rounding. This solution was tested against MPFR to confirm the results were correct.

The initial assumption would be that this would be true for all the 32-bit single input elementary functions, so it would be possible to simply cast to 32-bit and that would result in a nearly identically performant correctly rounded solution.

Next, it was noted that for many functions there is only a minimal performance difference between 32-bit and 64-bit mathematical functions when running on a modern 64-bit processor, in my case an AMD Ryzen 7 3750H. It is only the implementations that had significantly more complex 64-bit implementations that suffer a larger increase in run-time. This can be seen in particular for the implementation of Julia's `tan` function where the 32-bit implementation of the core kernel is the direct combination of a few simple polynomials (explaining the much higher rate of incorrect answers compared to other libraries in Table 4.2) whereas the 64-bit implementation is much more complex with more branching and special conditions. This results in the 64-bit implementation being nearly double the cost. However, consider that the `sin` function the 32-bit and 64-bit implementations are very similar and result in similar overall performance in Figure 5.1. It appears for implementations that are similarly implemented the 32-bit and 64-bit operations appear to be of approximately equal cost on modern 64-bit processors. A rewrite of the 64-bit implementations with this consideration may be able to reduce the cost of the 64-bit implementations when targeting 32-bit - it may even be possible to allow higher ULP error in the 64-bit implementation if it does not result in incorrect rounding when reduced to 32-bit.

5.3 Methodology

As with the last Chapter, this work was performed in Julia 1.4 [15]. For each of the functions, all positive floating-point inputs were analysed to determine the accuracy of the function.

For comparison, the GNU MPFR library was used because of its better support for testing 32-bit functions and so that comparisons of its results could be validated using higher-precisions. The GNU MPFR is also portable across operating systems whereas CR-LIBM is limited to Linux. Using a more portable library increases the reproducibility of this work so that it can be easier to externally validate.

Each 64-bit function is run and rounded to 32-bit, this is then compared against the MPFR library result. Any mismatch in the result is stored in a table with the input.

With the results, the new correctly rounded 32-bit functions are automatically generated in Julia. They are then tested using BenchmarkTools.jl [130].

Function	Mismatch	Function	Mismatch
sin	1	*sinh*	1
cos	2	*cosh*	0
tan	0	*tanh*	0
exp	0	*asin*	0
exp2	1	*acos*	2
exp10	0	*atan*	1
expm1	0	*asinh*	3
log	1	*atanh*	0
log2	0	*acosh*	2
log10	0		

Table 5.1: Number of incorrectly rounded results when casting the 64-bit Julia implementation to 32-bit for all positive 32-bit floating-point numbers.

5.4 Results

Table 5.1 shows that the maximum number of mismatches was three for `asinh`, with nearly half of all the functions tested having zero mismatches.

This means only a small number of functions needed to be specifically implemented to include the correct return value lookup to result in a fully correctly rounded result for the 32-bit implementations. The functions without mismatches can be directly rounded to 32-bit precision without loss of the correctly rounded property without any further intervention, when in Round-to-Nearest rounding mode.

5.5 Performance Evaluation

Looking at the performance results in Figure 5.1 shows that my performance is near identical to that of 64-bit implementations with a small cost for the branch — as would be expected.

There are some functions such as `tan`, `tanh` and `sinh` where this cost is significantly higher than the 32-bit implementation. For these functions a more specifically implemented correctly rounded solution may be more suitable or better performing, rather than taking on the significant performance cost of the 64-bit implementation. The implementation of a number of 64-bit functions are a lot more complex than the 32-bit counterparts and may explain the jump in the cost for a few functions which have been tested.

The relative performance of each implementation will vary on use and the CPU which is being used. This solution was benchmarked on an Intel i5 6600. If the targeted processor does not have a 64-bit supported FPU (Floating-point unit) then the relative cost of this approach will be higher.

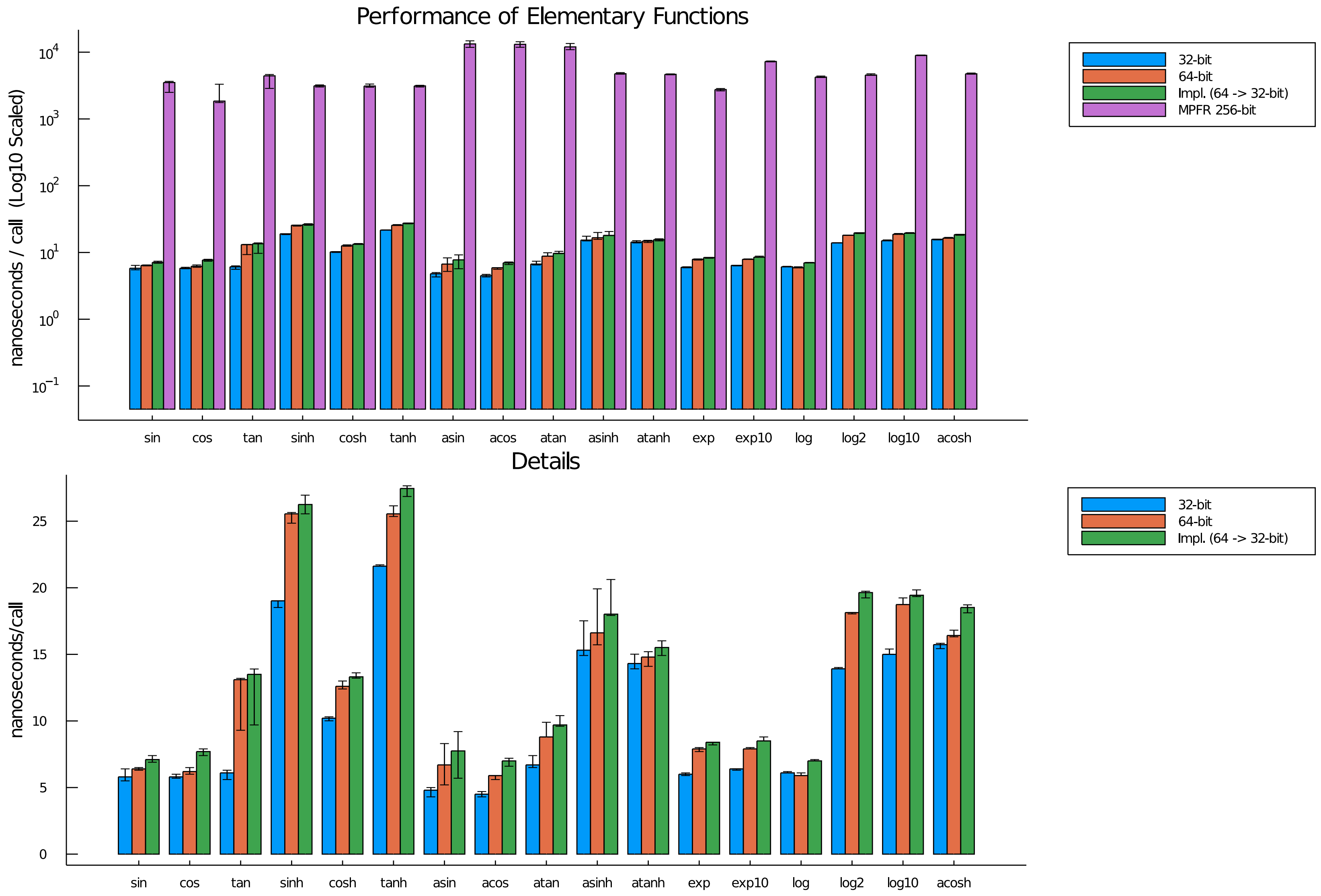


Figure 5.1: Performance of the Julia and MFPR elementary functions compared to the correctly rounded 32-bit method. Error is marked as the interquartile range. (Benchmarking tests performed on an *Intel Core i5-6600 CPU @ 3.30GHz*)

There is some credence to 32-bit floating-point and 64-bit floating-point having very similar performance characteristics during processing[2]. With the major downside to using 64-bit coming from having to double the amount of memory being used leading to less efficient cache behaviour. Although, this behaviour will vary greatly depending on individual CPU design. Figure 5.1 shows this in the relative difference in cost between the 32- and 64-bit implementation is for some implementations.

Others, such as `tan` have a significant difference in cost due to the significantly more complex 64-bit implementation. In the next section, it is shown how small 64-bit floating-point approximations can be used in these cases to produce a different 64-bit implementation with few incorrectly rounded results when cast back to 32-bit without a significant performance overhead.

5.6 Faster 64-bit `tan`

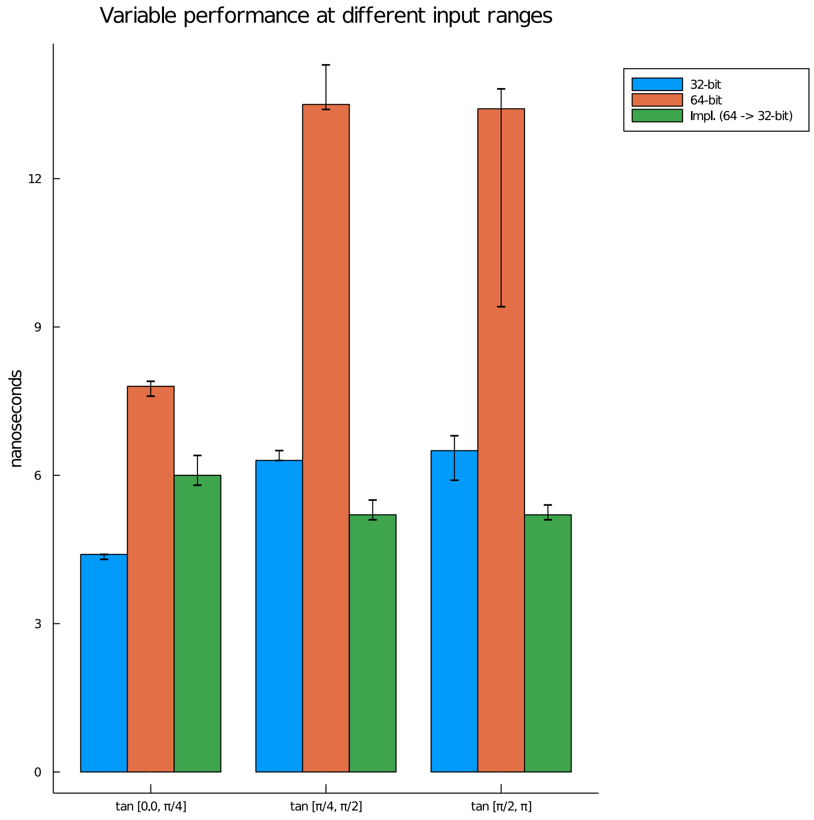
The implementations of `tan` between the 32- and 64-bit implementation vary greatly. The 64-bit implementation in Julia is very complex with many branching paths and complex mathematics to try and produce a good 64-bit result. This is how it managed to have zero incorrectly rounded results when it is cast down to 32-bit.

A result of this is that there is a large difference in the performance of the 64-bit implementation depending on which paths are taken due to their different complexity. This is visible in Figure 5.1 as the large error bars which are not seen in most other functions shown here. Figure 5.2 shows a more detailed breakdown of the different performance at different inputs. The paths taken for the lower gradient ranges of the `tan` function are much faster than the steeper sections which approach infinity.

The function's irregular performance and being a relatively expensive implementation when compared to the 32-bit implementation makes it less suitable for use in my proposed correctly rounded solution. To fix this, I present a Remez-based polynomial approximation replacement for the `tan_kernel` sub-function in the Julia implementation of `tan` which reduces the overall run-time of the function at the cost of some accuracy.

The new implementation, shown in Figure 5.2a, produces *fifteen* incorrectly rounded results when rounded down to 32-bit with a max ULP error of one. This is a significantly higher number of mismatches than the current 64-bit implementations which have been measured. There is a cost to using a table of many values or many if statements. Implementing my method directly results in function with a higher run-time than the existing 64-bit worst-case, even with a very cheap kernel. To fix this, a binary search approach is used when checking if the input is one of the known 15 incorrectly rounded results. With this design the new function is now approximately the same cost as the pure 32-bit implementation. The mismatched values are not

[2] Particularly moving to a 64-bit number from 32-bit on modern processors [143].



(a) Comparison of the run-time performance for different input ranges of the 32-, 64-bit and fast approximate 64-bit implementations of the `tan` function, with interquartile error. (Benchmarking tests performed on an *Intel Core i5-6600 CPU @ 3.30GHz*)

```
float32 fasttan64(float32 _x)
{
    // Get the range reduced value
    int     phase;
    float64 reducedval;

    phase,reducedval = range_reduce(_x);

    // Call the kernel with the correct input and phase.
    if iseven(n)
        return fastkernel(reducedval,1)
    else
        return fastkernel(reducedval,-1)
}


float32 fastkernel(float64 _x, int evenflag)
{
    // Remez rational polynomials
    static const float64[7] numerator   = [....]
    static const float64[7] denominator = [....]

    // Calculate the rational polynomial
    res = (numerator[0] + horner(_x, numerator[1:end]) )
        / (numerator[0] + horner(_x, numerator[1:end]) );

    if k == 1
        return float32(res)
    else
        return float32(-1.0/res)
}
```

(b) Pseudo-code of the implementation of the fast approximate 64-bit implementation. It uses the existing Julia range-reduction but approximates the inner `tan_kernel` with a rational polynomial.

Figure 5.2

evenly distributed, and as a result the function is more expensive in similar areas as the original 64-bit implementation but still significantly cheaper.

Using 64-bit approximations to produce correctly rounded 32-bit results can be investigated for many of the existing 64-bit implementations if the run-time difference between the 32- and 64-bit is too much of a performance penalty.

5.7 Conclusion

I have shown in this section it is possible to get well performing correctly rounded 32-bit solutions by taking advantage of the nearly-correctly rounded 64-bit solution.

For most functions the 64-bit implementation is of a similar cost, resulting in only a small overhead to get correct-rounding. In the cases where the cost of the 64-bit function is significantly higher than the 32-bit, I have shown it is possible to generate cheaper 64-implementations which are *good-enough* to produce only a small number of incorrectly rounded results when cast to 32-bit.

Overall this solution gives the same performance, exception and mathematical behaviours of the 64-bit implementation while maintaining the correct-rounded result in 32-bit.

For some functions, the jump from 32-bit to 64-bit implementations does not incur a significant performance penalty and therefore there is little reason to use the more incorrect 32-bit implementation for a lot of use-cases. For others, such as `tan` the function is 50% slower in 64-bit and therefore the change in performance may effect some programs if the faster, less accurate 64-implementation is not available.

In this chapter I have shown how to achieve correct-rounding for the tasks which require it. In the next chapter I will show how to get better performance while staying within the established error limits.

Chapter 6

Approximate `sin` to One ULP Max-Error

I have shown that existing mathematical libraries do not produce the correctly rounded result for many functions. This shows wide acceptance for error in single-precision floating-point functions, even though the latest IEEE-754 standard does not allow for any error in the results of these functions.

If, to the user, one ULP worst-case max-error is acceptable in all results from a mathematical function, then the existing strategy of aiming for correct-rounding but not reaching it for all results is not taking advantage of the optimisation space.

At the moment, the developers have created a situation where we must accept up to one ULP error for any result from the functions as a way to increase performance - but have not fully exploited that error towards the maximum performance possible for that level of error.

There likely exists transforms of existing mathematical software implementations which are less accurate than the existing solutions and provide better performance. Less accurate in the sense that more of the results are not correctly-rounded but no result exceeds one ULP max-error. If one of those transforms results in a smaller, cheaper or faster implementation of the function, then based on our survey of acceptable error in existing libraries, it should be seen as an overall acceptable improvement.

To explore the optimisation space of functions I developed the *ApproximateComputations.jl* package for Julia to help analyse, profile and produce approximations from existing functions. In this chapter, I use this library to transform the existing Julia `sin` implementation to allow for a greater number of one ULP error results as a means to increase the performance of the function. The key part of the library I will be using are the tools for sensitivity analysis.
For sensitivity analysis our library provides:

Min/Max value tracking to determine whether an instruction or set of instructions is actually a constant or gives the range of the values that can be represented;

Output fuzzing to identify by how much a value can deviate without the result of the function exceeding the known acceptable error boundary;

Automatic differentiation to assess the sensitivity of the output at positions in the input space;

Generalised code injection to insert custom probes at arbitrary points within the implementation.

The utilities provided are used to effectively map the sensitivity to error in the existing solution, determine where sensible changes could be made which are likely to not exceed the one ULP max-error limit and then verify any changes which are made.

This approach allows for small code changes in sensible places without having to brute-force test transforms on the entire function. I consider this to be a partial automatic approximation as the code can be written to automatically perform transforms where sensitivity is low, however, in this example the tools are shown being used in a supervised setup.

6.1 Sensitivity Analysis Overview

Not all operations in a function contribute equally to the final output of the function. Sensitivity analysis aims to identify the relative weighting of different operations on the final output of the functions, as well as considering how the output changes relative to those operations in different parts of the function domain.

As an example, in $f(x) = \mathrm{sqrt}(x) + 1$, $\mathrm{sqrt}(x)$ will have a diminishing impact on the final output as x approaches zero. Given an existing implementation of a floating-point function, we wish to identify appropriate places for approximation. These are sites where reduced precision has the smallest effect on the output of the function. When the error at these positions is less than is measurable at the working precision we are able to stop computing it.

When using *ApproximateComputations.jl* we start with a Julia compatible function as an expression list. This is then reprocessed into blocks that represent every possible sub-tree of the function.
As an example:

```julia
function example_function(x)
    10 + x * 2
end
```

Becomes:

Tree ID	Expression Tree
1	:(function example_function(x) 10 + x * 2 end)
2	:(example_function(x))
3	:example_function
4	:x
5	quote 10 + x * 2 end
6	:(10 + x * 2)
7	10
8	:(x * 2)
9	:x
10	2

This provides indices for injection points at any sub-expression tree in the function ranging from the whole function to individual variables or variable names. For the purposes of this explanation we elide all line numbers or other expression information which is not useful as injection points.

In the sample above, we can see that index one represents the whole function tree, indices 2-4 are the function header tree, and indices 5-10 represents the tree for the whole body of the function.

The library can remove, modify, or replace any individual expression. For example, to change or track the use of variable ‘:x’ in the above example we use the `NestTree` function at index 9 as follows:

```
function example_function(x)
    10 + x * 2
end
```

+

```
NestTree(targetFn, 9, :(TrackBlock))
```

↓

```
function example_function(x)
    10 + TrackBlock(x) * 2
end
```

The function `TrackBlock` is provided to change or track the value of `:x` at that position. The results can then be queried at the end of the run to determine the minimum, maximum or distribution of results for the value `:x` at that position. From this information, the library can plot the relative weighting of the variable compared to the output and map how it changes based on input.

It is then trivial to provide this wrapping for any code block in the application to produce a full mapping of the function.

With this we can easily and automatically inject code for fuzzing the output at different levels of the AST (Abstract-Syntax-Tree), which gives us the data we need to determine the sensitivity of every code-point.

6.2 Julia `sin` Implementation Analysis

In this section our tools are used to analyse the Julia `sin` implementation. This particular implementation of `sin` appears to share some core similarities to OpenLibm [86] which itself is derived from FDLibm 5.3 [151] and Sun Microsystems' msun (for single-precision[1]) [56], with changes for portability[2]. The wide reuse of these specific implementations shows that this is quite a common and widely used approach to implementing the `sin` function. Julia originally linked directly to OpenLibm but has since reimplemented many of those functions internally in the Julia language to gain the benefit from the JIT compiler and multiple-dispatch optimisations.

We now take a detailed look at the implementation before detailing the approximation analysis process.

The Julia implementation of `sin` is shown in Figure 6.1 and broadly consists of range-reduction followed by piece-wise polynomial approximation. Similar to the generic approach to implementing `sin` described in Chapter 3, the algorithm used in the function can be broken down into a number of steps for the input x which are annotated alongside the code in Figure 6.1:

1. **Check if the input $|x|$ is less than a very small number**

   If it is, then return this number. This is because close to zero, $sin(x) = x$

2. **Check if the input is within the acceptable domain of the sine function as given by the IEEE-754-2019 standard**

   This means x must be neither NaN nor ∞, and that x must be in the range $(-\infty, +\infty)$. If it is unacceptable, then return the appropriate error.

3. **Perform range-reduction**

[1] The implementation of `sin` in msun appears to diverge with FDLIBM with the 5.1 release in September 1993 and has been continually updated as part of FreeBSD.

[2] This includes extending the implementation to cover the type requirements of IEEE-754-2008 and Microsoft Windows compatibility.

As sine is a repeating pattern and floating-point calculations have better precision near zero, it is beneficial to try and calculate the value of x in it's smallest form. A full cycle of sine covers the range (0,2π) but due to symmetry only a quarter of this is required and so the range reduction changes the input to a value in the range (-$\frac{\pi}{4}$, $\frac{\pi}{4}$).

4. **Identify which quarter of the full cycle that *x* is present within**

   This determines how to transform the quarter-cycle value to position it appropriately.

5. **A piece-wise approximation**

   Approximation built from two polynomial functions. These two functions are called the `sin_kernel` (for the straighter section, Figure 6.2a) and the `cos_kernel`, Figure 6.2b (for the curve where the direction changes).

6. **Return the result**

The curve is separated into four quadrants. Each quadrant is represented by a curve constructed from either `cos_kernel` or `sin_kernel`. The `sin_kernel` sub-function represents the curve during its straighter sections, and the `cos_kernel` sub-function represents the curve during the part of the `sin` curve where the gradient changes direction. This piecewise separation allows for a section with a very small change or gradient to be represented separately to the area of the curve with the highest change of gradient. The source code for 32-bit floating-point `sin_kernel` and `cos_kernel` are shown in Figures 6.2a and 6.2b respectively.

As the current `sin` implementation is within one ULP of the correctly rounded result we can trust that the current range-reduction function `rem_pio2_kernel` is producing accurate results in the reduced space. We can also validate that the section of the algorithm responsible for the early return of the input when the input is very small to be correct. This leaves only the `sin_kernel` and `cos_kernel` functions as targets for approximation.

Before exploration can be started, limitations must be set on how this function can be changed. Without limitations, the approximation space is effectively infinite. The first limitation is that we do not wish to exceed the worst-case one ULP error. But the new approximate function is still aiming to otherwise be an IEEE-754 compliant `sin` function and therefore must also conform to the rules of an IEEE-754 `sin` function.

6.2.1 `sin` Constraints

The `sin`[3] function is a trigonometric function that is most commonly implemented based on the IEEE-754 standard [3]. In the latest revision of IEEE-754 `sin` is defined for a floating-point

[3] I refer to the `sin` function rather than the real sine function as I am referring to the specific function defined by the specific Julia 1.3 implementation. Not the real continuous, trigonometric function sine.

```
function sin(x::T) where T<:Union{Float32, Float64}
    absx = abs(x)
    if absx < T(pi)/4 #|x| ~<= pi/4, no need for reduction
        if absx < sqrt(eps(T)) # Step 1
            return x
        end
        return sin_kernel(x)    # Step 5 and 6 (opt.)
    elseif isnan(x)             # Step 2 (a)
        return T(NaN)
    elseif isinf(x)             # Step 2 (b)
        sin_domain_error(x)
    end
    n, y = rem_pio2_kernel(x)   # Step 3
    n = n&3                     # Step 4

    if n == 0
        return sin_kernel(y)    # Step 5(a)
    elseif n == 1
        return cos_kernel(y)    # Step 5(b)
    elseif n == 2
        return -sin_kernel(y)   # Step 5(c)
    else
        return -cos_kernel(y)   # Step 5(d)
    end
end
```

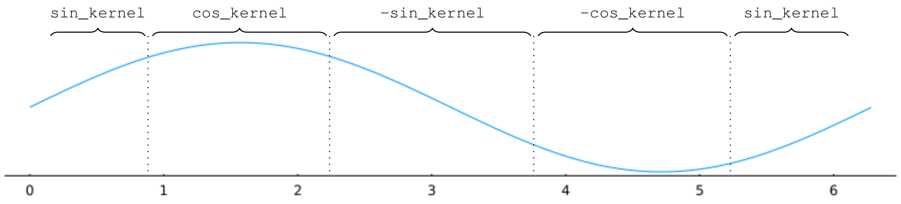


Figure 6.1: This code shows the base function used for `Float32` and `Float64` basic floating-point types in Julia. Annotated with step numbers to reference in the main text. The function `eps` is used which returns the step size from 1.0 to the next representable floating-point number for type `T` (where `T` may be `Float32` or `Float64` in this function).

```julia
@inline sin_kernel(x::Float32) = sin_kernel(DoubleFloat32(x))
@inline function sin_kernel(y::DoubleFloat32)
    S1 = -0.16666666641626524
    S2 = 0.008333329385889463
    z = y.hi*y.hi
    w = z*z
    r = @horner(z, -0.00019839334836096632, 2.718311493989822e-6)
    s = z*y.hi
    Float32((y.hi + s*@horner(z, S1, S2)) + s*w*r)
end
```

```julia
cos_kernel(x::Float32) = cos_kernel(DoubleFloat32(x))
@inline function cos_kernel(y::DoubleFloat32)
    C0 = -0.499999997251031
    C1 = 0.04166662332373906
    y2 = y.hi*y.hi
    y4 = y2*y2
    r = @horner(y2, -0.001388676377460993, 2.439044879627741e-5)
    Float32(((1.0+y2*C0) + y4*C1) + (y4*y2)*r)

end
```

sin_kernel(x)
0
1
2
3
4
5
6
cos_kernel(x)
0
1
2
3
4
5
6

(a) `sin_kernel`

(b) `cos_kernel`

Figure 6.2: `cos_kernel` and `sin_kernel` code and plots in range-reduced space. Note: `DoubleFloat32` is a confusingly named data-structure in the Julia maths package which holds two 32-bit floating-point numbers. Only one of these numbers is populated in this function.

input domain of values in the range $-\infty$ to $+\infty$ and should output the correctly rounded result as determined by the rounding mode.

In a previous chapter, it was shown that many implementations of this function do not produce the correctly rounded result. This results in incorrect rounding at different inputs in different implementations which means that the same input does not guarantee the same output between libraries. Many different implementations are used together — this is only possible due to a tolerance of the error in the difference between the two as they are not intensionally or Input-Output equal (see section 3.4 for more information on equality of functions).

The *IEEE-754 Standard for Floating-Point* states in its introduction that the standard will be provided 'for the development of common elementary functions'. The function `sin` is then defined in 'Section 9: Recommended Functions'. A brief summary of the requirements stated for `sin` to be conforming are:

1. The name given in a specific programming language does not need to match the exact name given by the standard. This is why `sin`, `sinf`, `sinl` and other variants are often seen, particularly in languages without multiple dispatch or function overloading.

2. A conforming operation shall return results correctly rounded for the applicable rounding direction for all operands in its domain.

3. Operation results shall be canonical.

4. A number of exceptions should be handled correctly (i.e. NaN results, NaNs in steps before the final result, division by zero, etc.)

5. `sin` will take the input domain of $(-\infty, +\infty)$

6. $|\infty|$ as an input is an exception. Underflow is an exception.

7. Inputs to `sin` is in radians (`sinpi` is a separate function which takes input in degrees and has its own requirements).

In summary, a conforming IEEE-754 defined `sin` function is to provide a mapping of the continuous sine function for $(-\infty, +\infty)$ into a supported floating-point standard value where the value is the correctly rounded result (for the current rounding-mode) of the true value for sine at that position. It will throw appropriate exceptions to inform the user of a problem with the input or internal processing and it is explicitly stated that the function will not do any work that is not the minimal needed to achieve the final result.

From this definition of constraints, all the restraints that are currently being followed in the current implementation are being taken.

```
"""
    @horner(x, p...)
Evaluate `p[1] + x * (p[2] + x * (....))`, i.e. a polynomial via Horner's rule.
"""
macro horner(x, p...)
     xesc, pesc = esc(x), esc.(p)
    :(invoke(evalpoly, Tuple{Any, Tuple}, $xesc, ($(pesc...),)))
end

# Evaluate p[1] + z*p[2] + z^2*p[3] + ... + z^(n-1)*p[n].  This uses
# Horner's method if z is real, but for complex z it uses a more
# efficient algorithm described in Knuth, TAOCP vol. 2, section 4.6.4,
# equation (3).
```

Figure 6.3: `horner` macro from ‘Julia/base/math.jl’

6.3 Analysing the Sensitivity of the Existing `sin` Implementation

I will now demonstrate the use of `ApproximateComputations.jl` to analyse the subfunctions of the Julia 1.3 `sin` function implementation.

To approximate a function it is necessary to be able to analyse what is happening at each stage and determine where a change would result in better performance. This is an exploration of the optimisation space for this specific implementation. This helps identify if any suitable better performing changes exist which result in an acceptable error.

In the last section, we identified the `sin_kernel` and `cos_kernel` to be suitable subfunctions to begin to approximate as the higher-level code in the function would be unlikely to result in any changes which would be beneficial.

6.3.1 Setting Up Sensitivity Analysis of `sin`

The sub functions `sin_kernel` and `cos_kernel` make use of a macro called `@horner` which we first need to expand.

The `@horner` code listed in Figure 6.3 shows how the macro is expanded. It produces a polynomial of the coefficients in Horner form [25]. Horner form is used because it reduces the needs to compute and store each subsequent power of the input value. In practice, this results in a chain of fused-multiply-add operations for the number of inputs. Applying the macro in the kernel by hand to our `sin_kernel` gives the output shown in Figure 6.4.

After applying the macro the function is only dealing with basic operations (+,-,*,/). The function is also in SSA (single static assignment) form [34] which simplifies analysis by not having conflicting data assigned to the same variable name. With the `@horner` macro operations unrolled it is possible to proceed with analysing the behaviour of each step of this function for its input domain.

```
function sin_kernel_unrolled(y::Float64)
    S1   = -0.16666666641626524
    S2   = 0.008333329385889463
    z    = y*y
    w    = z*z
    s    = z*y
    ex   = 2.718311493989822e-6
    r    = z * ex + -0.00019839334836096632
    sres = z * S2 + S1

    Float32((y + s*sres) + s*w*r)
end
```

Figure 6.4: Unrolled sin_kernel

For this function, we are interested in the ranges of values produced for each step in the function and their sensitivity in affecting the final returned value.

For example, if we had the function $f(x) = x^2 + 1$ which has an input domain of $[1, 5]$ then the min and max values of the code-point x^2 would be $[1, 25]$ and for the code-point at $+1$ the min and max would be 1. When we look at the sensitivity we would see that $+1$ is linear, a small change of 1 results in an equal change of output. But for x^2 we would find that fuzzing the value of x would result in a larger change of final result when x is larger due to its non-linearity.

The minimum and maximum range of the code-points are important in identifying constants or values which do not change very much. And we care about the sensitivity of each code-point so that we can identify if, and where, in the input domain that a code-point has a significant impact on the final result.

When we find a variable or code-section that is constant or near-constant we can then understand what affect changing it to be a fixed constant would have on the final output. This is profile-guided redundant code elimination.

With the *ApproximateComputations.jl* library, both of these tests are trivial to implement. The first is implemented by injecting code to track the value of each branch in the AST and storing the index of that branch and the two associated values. This can be extended to store all the values that are returned to get the distribution of returned values compared to the input. The fuzzing is implemented by injecting code to modify the return value of each AST branch, one at a time, by a given distribution of fuzzing. The result of the fuzzed implementation is compared to the original implementation and the difference can be used to plot a 3D graph of input, difference and fuzz value, which can show whether the sensitivity is large or small, or if the sensitivity is only high or low in certain input ranges.

As we are dealing with single-input 32-bit floating-point functions it is possible to take a brute-force approach when profiling the functions we are interested in. More complex functions or those using larger precise data-types would need a distributed approach, such as Monte-Carlo sampling, to gain insight into the range sensitivity spaces in finite time.

```
function sintest(x::T) where T<:Union{Float32, Float64}
    absx = abs(x)
    if absx < T(pi)/4 #|x| ~<= pi/4, no need for reduction
        if absx < sqrt(eps(T))
            return T(NaN)
        end
        return sin_kernel_base(x)
    elseif isnan(x)
        return T(NaN)
    elseif isinf(x)
        sin_domain_error(x)
    end
    n, y = Math.rem_pio2_kernel(x)
    y = Float64(y.hi)
    n = n&3
    if n == 0
        return sin_kernel_base(y)
    elseif n == 1
        return T(NaN)
    elseif n == 2
        return -sin_kernel_base(y)
    else
        return T(NaN)
    end
end
```

Figure 6.5: sin_kernel only sin function.

6.3.2 Performing Sensitivity Analysis

It was shown in Chapter 4, that the majority of incorrectly rounded results are in the input domain which is handled by the `cos_kernel` quadrants of the curve. This means that in the `sin_kernel` domain there are many more values which could be exploited by decreasing the accuracy to be closer to one ULP error.

When analysing the `sin_kernel` function alone, we want to only analyse the inputs from `sin` which go down a path involving `sin_kernel`. We can change the base sin function to return `NaN` when not going down a path to the `sin_kernel` function. This modification, shown in Figure 6.5, reduces redundant calculations.

We start our analysis of this reduced version of `sin` by plotting a histogram of the ULP error for all inputs within the valid domain of the sub-function `sin_kernel` to explore the error distribution of the existing implementation. Figure 6.6 shows the error distribution and it is clear that the vast majority of this sub-function is correctly rounded with only a small number of values which would round to 1 ULP of error. This is expected and is why we are primarily looking at the `sin_kernel`. It has the most correctly rounded values to exploit.

When the `sin_kernel` function is indexed by the approximation library, it finds 71 points of the AST which are valid attachment points for code injection or measurement.

If the list of attachment points is filtered to remove constant variables, constructors, operators and large blocks, a much more manageable list is produced. We avoid measuring these because they are unlikely to be able to be trivially changed to gain much performance without breaking

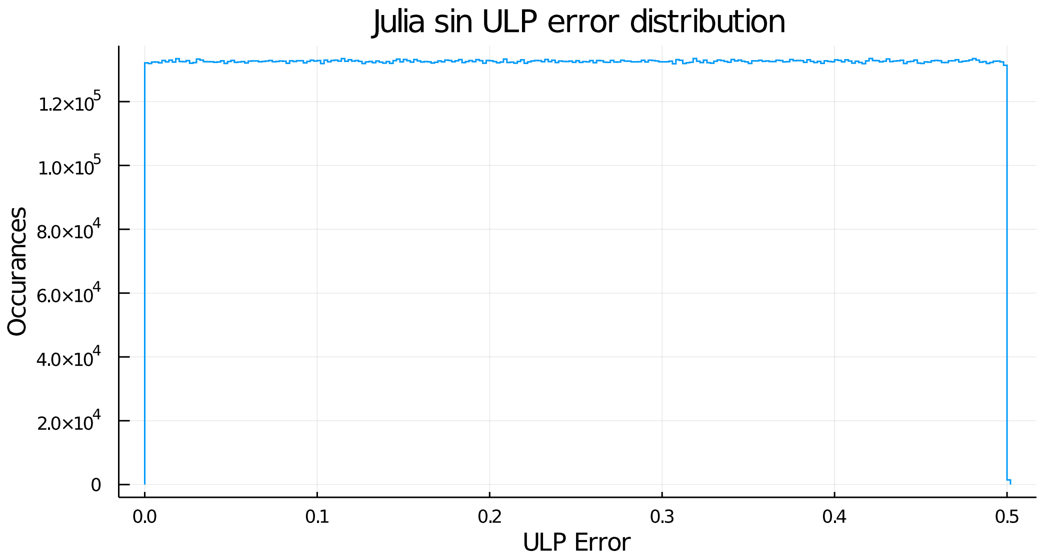


Figure 6.6: ULP error distribution of a sin_kernel only sin function. Note the small number of elements with > 0.5 ULP error on the bottom right.

the one ULP max-error limit. The list below shows the injection index and corresponding code-block for likely optimisation candidates. It shows that we can instrument whole lines or simply small sections of that line.

Index	Code-block
2	$sin\_kernel(y :: Float64)$
14	$z = y * y$
20	$w = z * z$
26	$s = z * y$
37	$z * ex + -0.00019839334836096632$
49	$z * sresx + S1$
60	$y + s * sres$
35	$r = z * ex + -0.00019839334836096632$
47	$sres = z * sresx + S1$
58	$(y + s * sres) + s * w * r$
56	$Float32((y + s * sres) + s * w * r))$
67	$(s * w * r)$
⋮	⋮

To start by tracking all the assignments to local variables we can make use of the `TrackLocal()` function which will take advantage of the function being in SSA form. `TrackLocal()` when called will store a snapshot of the current value of every locally declared variable in a function so that we can access it later.

We are not interested in the return value as we already have that information tracked from calling the function. So we can use the attachment point 56 which is the point before the final return line of the function[4] and inject our `TrackLocal()` function there to find the minimum and maximum values for any variables in the function after it has been executed. That produces this min-max information for each assignment in the function:

VarName	Min	Max
:w	1.1781e-15	0.376234
:s	-0.48039	0.470575
:S2	0.00833333	0.00833333
:S1	-0.166667	-0.166667
:y	-0.783185	0.777815
:z	3.43235e-8	0.613379
:ex	2.71831e-6	2.71831e-6
:r	-0.000198393	-0.000196726
:sres	-0.166667	-0.161555

From this table we can see that `:r` and `:sres` vary by only a very little across the full domain of the function. This makes them near constant, and therefore the operations which assign to them are a very interesting target for optimisation:

VarName	Min	Max	Code
:r	-0.000198393	-0.000196726	z * ex + -0.000198393…
:sres	-0.166667	-0.161555	z * s2 + S1

Instrumenting the assignments to `:r` and `:sres` to record the value assigned for each valid input in the domain gives the plots shown in Figure 6.7. The table shows that `:r` has a very tiny range, and `:sres` also only changes in value by a small amount but with a magnitude much larger than `:r`. This represents a range of 114556 floating-point steps for $: r$ and 343061 for `:sres`.

This shows that `:r` would be the most likely value to have little impact on the final result. Removing accuracy from this line of code is more likely to produce a smaller change of result and the number of possible values covered by this variable is smaller. Therefore, a change is less likely to break our limit of max one ULP error.

The values of `:r` are not so small as to be destructively combined with larger floating-point values within this function[5], but the binade of the ranges for values expressed by $: r$ are 4 times

[4] In Julia if no return is explicitly stated the result of the last line of the function is returned.

[5] The smallest binade with a step-size equal to or greater than the value of $: r$ does not begin until 2048.0f.

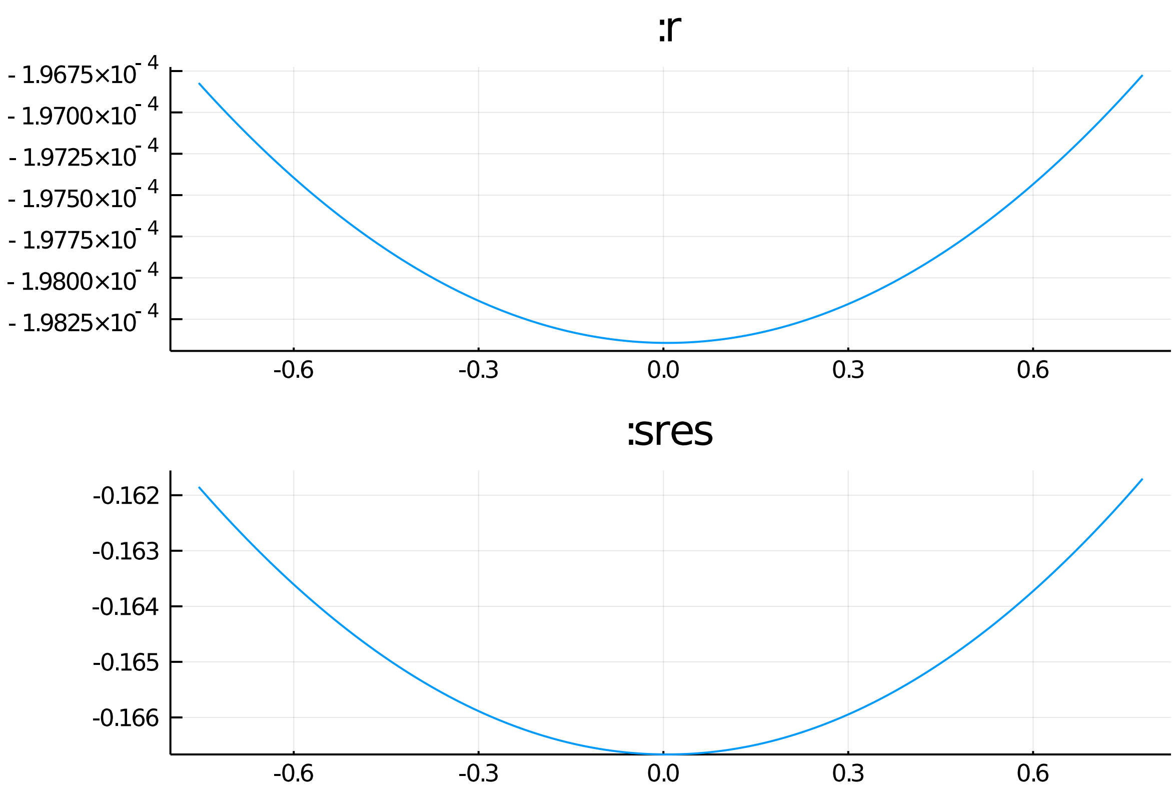


Figure 6.7: The results for `:r` and `:res` for inputs within their domain. They both represent a simple and similar curve, but with a different scale.

smaller than the binade of the largest values it is seen in combination with, so some value loss is highly likely for some input values.

The variable `:r` represents a scaling factor on the `:s` and `:w` variables before they are added to `(y + s * sres)` in the final line of the function. From our analysis of the ranges of `:w` and `:s`, it can been seen that `:r` is scaling what is sometimes a very small number. As such, it is possible that a small change to the calculation in the already very small value of `:r` may result in a viable optimisation that may not significantly change many results.

6.4 Applying an Optimisation

In the last section, we identified that the computation `r = z * ex + -0.00019...` has the potential for optimisation to reduce the number of operations while maintaining an overall max error of one ULP for the sin function.

The line of code uses a single addition and multiplication, reads one variable, and uses two constants. Our optimisation options are:

1. Using an existing calculation within the function to replace some or all of the variables in this line. E.g replacing `:(z*ex)` with `:s` if it was thought that `:s` holds approximately similar values across the function domain.

2. Remove parts of this line of code. Such as removing :`(z*ex)` because the computation may be so small as to not be significant to the final result.

3. Or, replace the whole calculation with a constant. E.g :`(r = 0.1)` if the constant gives good enough results for inputs in the domain.

For replacing with existing values within the function, we are a little out of luck as no other variable is at the correct scale so we would still need to apply the scaling factor from :`ex` which would not save a multiplication.

If we wanted to remove sections of the function it may be possible to remove the final subtraction from the line which assigns :`r`. That constant only represents 3% of the total value space covered by :`r`, which may not be significant enough to result in a large enough error when combined with other variables in the final line of the function.

Finally, we may select a constant.

This gives us two tasks that are sensible approaches to investigate to see if they may produce a valid result. Firstly, remove the subtraction. Secondly, identify if any constant would suffice.

The first approach would save on one subtraction and a constant read. The second would save on a multiplication as well as the subtraction and constant read. We would like to say that the constant would remove the need for the ':z' variable but unfortunately, that variable is used again in the second last line of the function.

6.4.1 Removing a Subtraction

When the constant was removed approximately 70% of the test values did not match with the correctly rounded result, and further, when the full range of inputs was tested the overall worst-case ULP was over 600.

This makes this simple change unacceptable. The only remaining optimisation option is to find a constant to represent `z * ex + -0.000198393...` which will meet the requirements.

6.4.2 Using a Constant

We now instrument the function to set the value of :`r` and then search for a constant value to place there that results in less than one ULP max error.

From our analysis, we know the minimum and maximum value that :`r` is ever assigned: -0.000198393 to -0.000196726. We assume that if an optimal constant exists it should be within this range of 114556 32-bit floating-point numbers, of which only 1562 values are ever produced by the function normally.

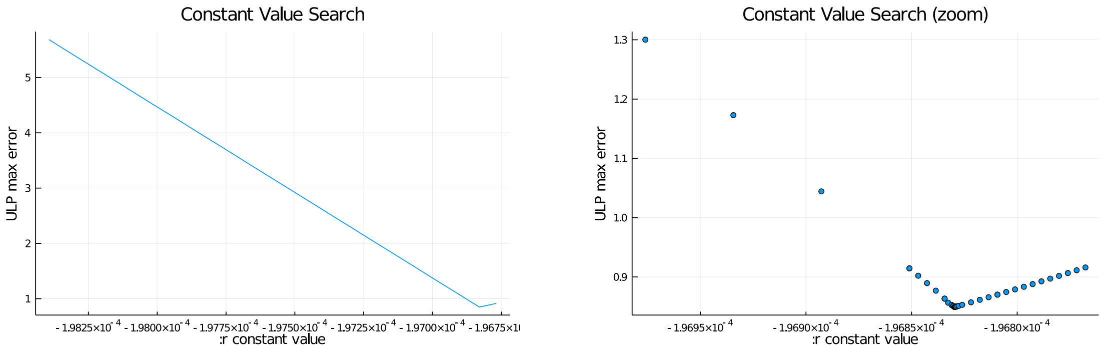


Figure 6.8: Descending search for the correct constant replacement for the `sin_kernel`.

```
function sin_kernel(y)
    S1 = -0.16666666641626524
    S2 = 0.008333329385889463
    z = y * y
    w = z * z
    s = z * y
    r = -0.00019682967f0        % Changed from z*ex + -0.00019839334836096632
                                % f0 signifies a 32-bit floating-point number.
    sres = z * S2 + S1
    Float32((y + s * sres) + s * w * r)
end
```

Figure 6.9: Optimised sin_kernel where the calculations for `:r` have been replaced with the constant -0.00019682967f0.

Our initial search revealed that there is likely a single minimum. This minimum was converged upon using gradient descent to find the constant which produced the minimum worst-case ULP error. Figure 6.8 shows the results of our search.

This search provides us with the constant value of -0.00019682967 which gives a worst-case ULP error of 0.84924996 for the whole domain, safely less than 1 ULP error. Figure 6.10 shows the distribution of the ULP values for the range of the `sin_kernel` function compared to new modified `sin_kernel` with the constant. The distributions of values have changed and now there is a larger number of results greater than 0.5ULP error but none which are greater than one ULP when compared to the correctly rounded high-precision result. Inserting this constant into the original `sin_kernel` gives the source code shown in Figure 6.9 with a ULP error distribution shown in Figure 6.10.

To confirm that this is an improvement, by the metric of fewer instructions, we need to look at the code generated. Figure 6.11 shows the comparison between the original `sin_kernel` and the new optimised version, where we use fewer instructions after the change.

Simply having fewer instructions is not always a sign that a function may have better performance, however, in this instance we have only removed existing instructions and not replaced them with more complex instructions, or instructions which would be less performant.

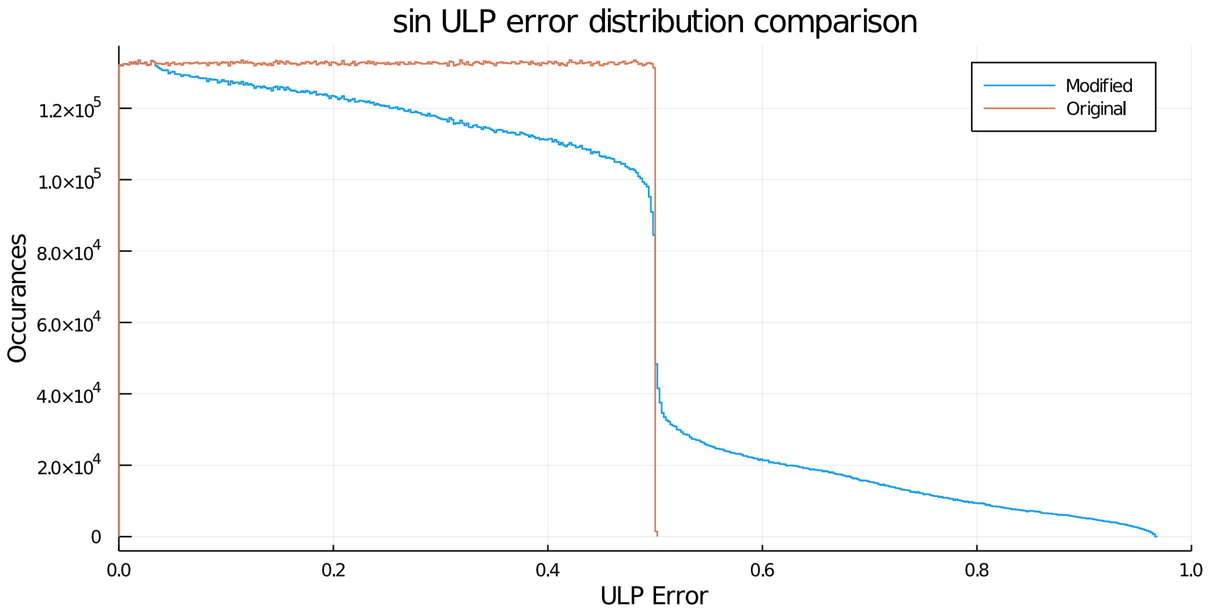


Figure 6.10: Showing the redistributed error when the function is modified to take a constant in place of the multiplication and additions.

In benchmark comparison (Figure 6.12) we see that there is an 11% performance gain when using the optimised one-ULP solution.

6.4.3 Results

In our search and testing of the best replacement constant, we have found a large number of possible values which would result in a ULP max error which is less than one, and we have found the 32-bit floating-point constant which results in the lowest max ULP.

To conclude the work on the sin function, we have found that for implementations based on the common FDLIBM/msun implementation of the `sin` function it is possible to remove some operations and end up with a function within the same tolerances as is already implemented.

6.5 Conclusion

The current implementation of some mathematical functions accept a one ULP max error bound. It has been shown here that is it possible to optimise towards this bound, rather than towards correct-rounding to create a more optimal function within the same error tolerances.

Ideally, values should only be computed up to the precision necessary to perform the task needed. As this can often be unknown, the target accuracy is so often perfect accuracy as expressed through correct-rounding. However, when that cannot or should not be achieved for performance reasons then the true max error bound should be the target. It would be beneficial to some systems in the future to be able to apply the methods shown here automatically to compile a written function into a more optimal form for given error parameters and input domains.

```
pushq    %rbp
movq     %rsp, %rbp
vmulss   %xmm0, %xmm0, %xmm1
vmulss   %xmm1, %xmm1, %xmm2
vcvtss2sd        %xmm1, %xmm1, %xmm3
movabsq  $420550232, %rax
vmulsd   (%rax), %xmm3, %xmm3
movabsq  $420550240, %rax
vaddsd   (%rax), %xmm3, %xmm3
movabsq  $420550248, %rax
vmulss   (%rax), %xmm1, %xmm4
movabsq  $420550252, %rax
vaddss   (%rax), %xmm4, %xmm4
vmulss   %xmm0, %xmm1, %xmm1
vmulss   %xmm4, %xmm1, %xmm4
vaddss   %xmm0, %xmm4, %xmm0
vmulss   %xmm2, %xmm1, %xmm1
vcvtss2sd        %xmm1, %xmm1, %xmm1
vmulsd   %xmm1, %xmm3, %xmm1
vcvtss2sd        %xmm0, %xmm0, %xmm0
vaddsd   %xmm0, %xmm1, %xmm0
vcvtsd2ss        %xmm0, %xmm0, %xmm0
popq     %rbp
retq
nop
```

(a) Original function.

```
pushq    %rbp
movq     %rsp, %rbp
vmulss   %xmm0, %xmm0, %xmm1
vmulss   %xmm1, %xmm1, %xmm2
vmulss   %xmm0, %xmm1, %xmm3
movabsq  $420550364, %rax
vmulss   (%rax), %xmm1, %xmm1
movabsq  $420550368, %rax
vaddss   (%rax), %xmm1, %xmm1
vmulss   %xmm1, %xmm3, %xmm1
vaddss   %xmm0, %xmm1, %xmm0
vmulss   %xmm2, %xmm3, %xmm1
movabsq  $420550372, %rax
vmulss   (%rax), %xmm1, %xmm1
vaddss   %xmm1, %xmm0, %xmm0
popq     %rbp
retq
nopl     (%rax)






.
```

(b) Optimised function.

Figure 6.11: Generated assembly for the original and optimised function.

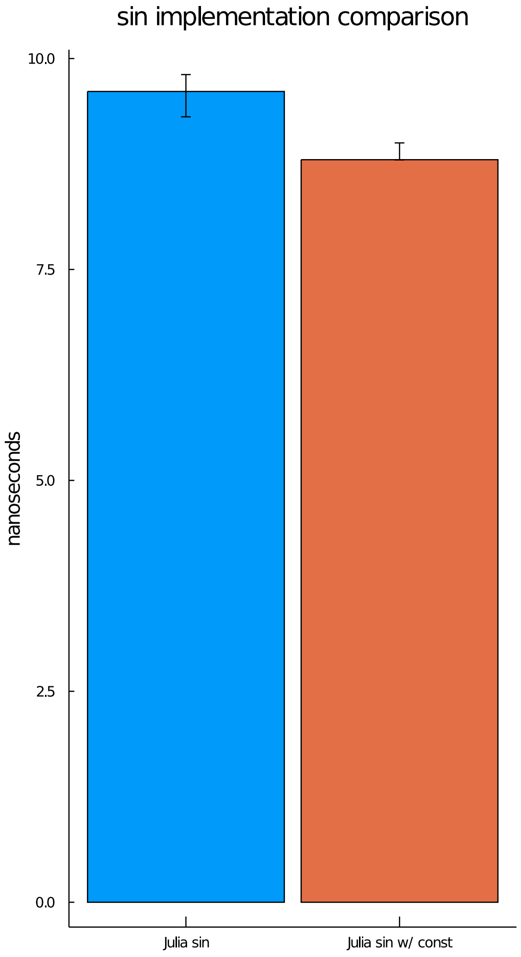


Figure 6.12: Comparison of the performance of the source Julia `sin` function and our approximation which replaced an expression with a suitable constant. Benchmark ran on an *Intel i5 6600* for inputs in the range which use the `sin_kernel` sub-function, using BenchmarkTools.jl in Julia.

I have demonstrated the use of supervised automated search techniques for finding these types of approximations and hope to develop this further to streamline the library to allow for fully automated optimisation in the future[6].

In the next chapter it is shown that further performance gains are available if applications are tolerate to even higher levels of error.

[6] Particularly with the increase in support for code-level automatic differentiation in the latest Julia releases.

Chapter 7

Case Study: Neural Network Activation Functions

In the previous chapters I have shown the existing problems with the elementary functions used in standard libraries and shown how functions of differing accuracy can be produced to match the error requirements of a task. In this chapter I demonstrate how selecting an approximate mathematical solution for error tolerant applications can result in useful performance improvements without impacting the quality of the final result. For this case study *Neural Networks* are used as the target program due to their inherent error-tolerance, and specifically targeting the simple mathematical functions used as activation functions between layers.

Neural networks rely on activation functions to provide non-linearity to its training process but they are not constrained by having to be precise calculations. *Recurrent neural networks*, such as LSTMs (long short-term memory) make extensive use of both the hyperbolic tangent and sigmoid functions. The hyperbolic function is a standard elementary function which have been profiled in earlier chapters, it was seen that it is one of the commonly more computationally expensive functions of this set. The sigmoid function is reliant on the function `exp`, another elementary function.

I used function approximation techniques to develop replacements for these functions and evaluated them empirically on three popular network configurations. This chapter shows safe approximations that yield a 10% to 37% improvement in training times on the CPU. These approximations were suitable for all cases that were considered, and I believe are appropriate replacements for all networks using these activation functions. Ranged approximations were also developed which only apply in some cases due to restrictions on their input domain. The ranged approximations yield a performance improvement of 20% to 53% in network training time. The functions also match or considerably out-perform the ad-hoc approximations used in Theano [13] and the standard reference implementation of Word2Vec [111]. This chapter shows the application of my work in general approximation of mathematical functions to arbitrary levels of precision in Chapter 6 applied towards the mathematical functions used in the activation functions of neural networks.

Neural networks are not only inherently tolerant to error but practitioners will admit that there is a certain art to the selection of activation functions (and other hyper-parameters) and thus a certain leeway in their accuracy.

Function approximation has been applied to activation functions before. The implementation of Word2Vec makes use of a lookup table for approximating the exponential function, and the popular machine learning library Theano included an approximation of the `sigm` function called `ultra_fast_sigmoid`. Google also released some research where they showed a piecewise approximations of their 'Swish' activation function for use in their edge computing TensorFlow interface which shows the viability of some approximation for limited hardware [81]. However, this work is the first detailed study on the subject in this area, and shows that approximations can be used as drop-in replacements for current models. The approximations presented outperform all of the mentioned alternative approximation approaches.

In this chapter a range of approximations are considered with a trade-off between complexity and accuracy. The overall impact on training and prediction time for popular neural network architectures are studied. I identify two approximation options for each: a *safe* approximation that works in all cases and a *ranged* approximation that requires some restrictions to it's input domain. For all networks tested it is found that my safe approximations outperform the standard implementations with an 10% to 37% improvement in training times. Where appropriate the ranged implementations provide an improvement of 20% to 53%. The training and inference benchmarks are initially run on CPU as a proof of concept. The performance benefit on GPU will be different, especially when using the standard elementary functions which are already optimised for performance over total correct-rounding accuracy as shown in Chapter 4.

I believe that the *safe* approximations are of particular relevance to neural network library developers since they are better-performing drop-in replacements for existing functions. There is also a growing range of specialised hardware designed for accelerating machine learning tasks and in future work I would like to consider how well my approximations might translate into hardware implementations.

An open-source implementation of my work is provided, FastApproximations.jl[1], in Julia for use with the Flux [83] machine learning library. I argue that these approximations are applicable to other libraries too and a brief study using TensorFlow [5] is included as evidence.

7.1 Activation Functions

Activation functions are used in neural networks to introduce non-linearity. For activation functions with bounded results, the popular choices are logistic, trigonometric and clamping

[1] Available at: `https://github.com/NTimmons/FastActivations.jl`

functions with the sigmoid (`sigm`) and hyperbolic tangent (`tanh`) functions being used most frequently. For unbounded or partially bounded activations, the `ReLU` function and it's variants are most popular.

`ReLU` is the dominant choice for an activation function but `sigm` and `tanh` still have significant use as gating functions in feed-forward and recurrent networks.

I focus on `sigm` and `tanh` in particular as they are major components of an LSTMs gating behaviour, which in it's original form uses three calls to `sigm` and two calls to `tanh` [73].

For cases where `ReLU` is not appropriate (such as LSTMs) the performance impact can be notable: in micro-benchmarks measuring function execution time I found `sigm` and `tanh` to be 3.6x and 7.9x more expensive than `ReLU`. This chapter focuses on these two functions given their popularity and relatively high cost.

A significant proportion of CPU time is spent in the computation of values in activation functions. For example, for a densely connected network with 51,000 trainable parameters using a `sigm` activation function with two equal-sized hidden layers it was measured that approximately 29% of time is spent computing the `sigm` function. For context, in the paper which introduced sequence-to-sequence learning for neural networks the authors use a 5-layer LSTM network which has 380 million trainable parameters [153].

7.2 Activation Function Approximation

A conventional implementation of a mathematical function will aim to compute results which arise from rounding actual values to the precision of the data-type used. An application might be able to tolerate considerably more error than this but it can be difficult to prove conclusively. In this chapter we seek to validate my approximations empirically. We believe this is appropriate since the 'correct' choice of activation function is most often demonstrated empirically too.

In addition to tolerating error in the function's outputs an application might also restrict the function's input domain. This provides further opportunity for approximation since it is only necessary to mimic the original function within the domain of interest. We therefore develop two classes of approximation. The *ranged* approximation produces higher performance under the assumption that the input domain is approximately -5.5 to 5.5 [2]. The *safe* approximation makes no assumptions about input domain and so perform as a drop-in replacement for all cases. They do not exceed the output bounds of the original function.

Two different approximation approaches are considered: 1) taking a mathematically derived approximation of the function and then using the parameters to produce a range of alternative versions at varying precision; 2) performing a numerical regression to fit the given function. The mathematically defined approach produces more stable results but can be more complex. The

[2] These values are where the `sigmoid` function levels off at the minimum output of 0.0 and the maximum output of 1.0.

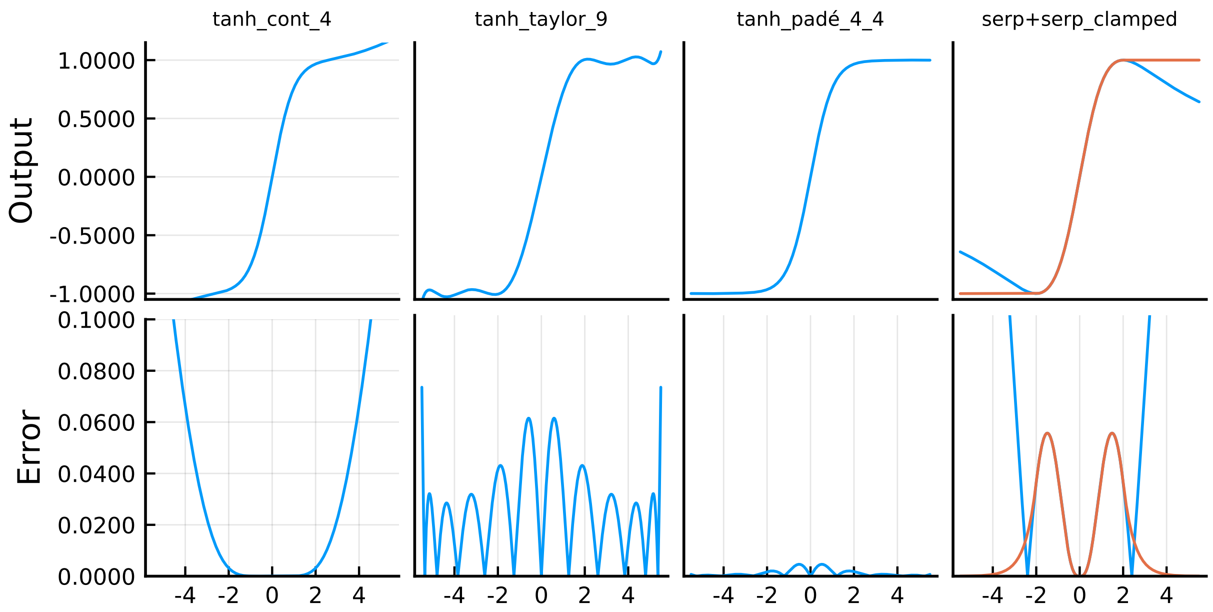


Figure 7.1: Approximation functions for `tanh` and their associated error calculated as the absolute difference between the original `tanh` function per input x.

regression approach produces often faster and simpler functions but the output result is heavily determined by the input parameters and function that is being replaced.

This section covers what each activation function is and then a proposal on how to optimise it, both safely and unsafely.

7.2.1 `tanh`

`tanh` is the function to compute the elementary hyperbolic tangent function. The output is an 'S-curve' in the range -1.0 to 1.0. It is defined as:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Four forms of approximation of `tanh` are considered which are shown in Figure 7.1 and describe below.

Firstly `tanh` is implemented as Lambert's continued fraction. I limit the implementation to a finite number (n) of iterations depending on the precision desired and then simplify using a symbolic algebra package. I call this approximation `tanh_cont_n`.

$$\frac{x}{1 + \frac{x^2}{3 + \frac{x^2}{5 + \dots}}}$$

Two forms of polynomial approximation of `tanh` are also considered: 1) a Taylor Series of the form $\sum_{i=0}^{n} a_i x^i$ and; 2) Padé approximants of the form $\frac{\sum_{i=0}^{n} a_i x^i}{\sum_{i=0}^{m} b_i x^i}$. A desired number of terms

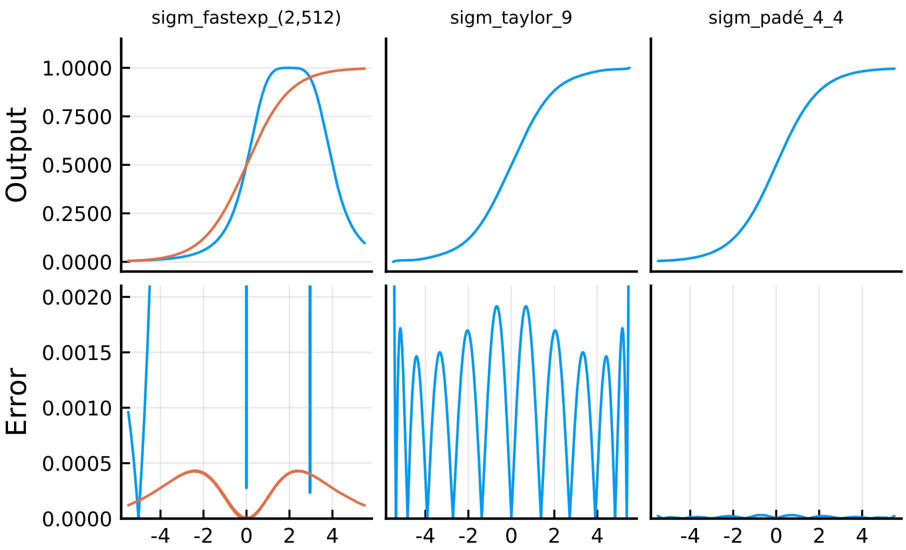


Figure 7.2: Approximation functions for `sigm` and their associated error calculated as the absolute difference between the original `sigm` function per input x.

(n, m) are selected, choose a uniform sample of 5000 values in the range -5.5 to 5.5 and then determine the values of the coefficients (a_i, b_i) using the Remez Algorithm.

Using these two techniques yields the approximations `tanh_taylor_n` and `tanh_pade_n_m`. Variations in the number of points sampled and the range covered during fitting of these functions did not have a big impact on the results.

As a final option for `tanh` the Serpentine function (`serp`) is considered and defined as:

$$y = \frac{2x}{x^2+4}$$

As the `serp` function does not fit `tanh` beyond the main gradient, a variant `serp_clamp` is also introduced which is clamped to the values -1 or 1 for inputs outside of the range -2 to 2.

7.2.2 `sigm`

The sigmoid function is an S-curve defined as:

$$\texttt{sigm}(x) = \frac{1}{1+\texttt{exp}(-x)}$$

Figure 7.2 shows three approximation forms for the `sigm` function. These arise from the approximation of the `exp` function included within the `sigm` implementation. Here a well-known expansion of the `exp` implementation is shown for an infinite limit.

This is implemented as a function which takes a target, n, for the total number of iterations it should approximate to. Through multiplication rules it reaches the target n in $\log(n)$ multiplications which significantly reduces the computational cost of higher iteration counts.

I call approximations of this form `sigmoid_fastexp_n`.

$$\texttt{sigm}(x) = \frac{1}{1+\texttt{exp}(-x)} \qquad \texttt{exp}(x,n) = \big(1+\frac{x}{n}\big)^n$$

I also generate fits for Taylor and Padé polynomials in the same manner as for `tanh`. This yields approximations `sigmoid_taylor_n` and `sigmoid_pade_n_m`.

Note that there is an alternative approach to the fast computation of the exponential function based on exploiting the definition of IEEE Floating-Point numbers [142]. This is no longer as effective as it once was due to it's reliance on a union structure to treat the floating-point number as both an integer for some parts and a floating-point number for others. This technique is not easily transferable to SIMD [120] where arrays of numbers are worked on in parallel and it is not always trivial to cast between different type representations in memory.

7.2.3 `ReLU`

`ReLU` stands for Rectified Linear Unit (ReLU). This is a very complicated name for a simple function. `ReLU` is a linear function which returns 0.0 if the input is below zero.

$$\text{relu}(x) = max(x, 0.0)$$

The first consideration for an optimisation of `ReLU` is one which transforms a max operator (which may use a branch) to a sum and a division.

$$\text{relu}(x) = (x + |x|) * 0.5$$

This may result in changing the return value from x to a value near to x under the rules of IEEE floating-point arithmetic [1]. The error would come from the multiplication by 0.5 resulting in a number which could not be accurately represented at the given precision. This optimisation trades on this accuracy to avoid the branch in the if-statement of the `max` function. This is unlikely to result in a significant increase in performance due to fast comparison hardware common in many modern CPUs, but demonstrates a simple transform to avoid a potential slowdown on some hardware. In my tests this resulted in an insignificant change and so the standard implementation of `ReLU` are used as the baseline.

7.3 Performance Results

In practice it is the performance of the whole network which is important rather than a particular activation function. For example, when training a neural network the ideal choice of activation function will result in the lowest loss[3] for the least training time. This means that when selecting an approximation we are looking for a trade-off between the computation cost and the resulting error. Fast approximations with high error might cause a network to take longer to converge than slower approximations with a lower error.

Earlier in this dissertation an example was given where using library implementations which were not correctly rounded resulted in networks training towards different values. It is important to state here that in this test we are not looking for conformance between the different activation functions used and are instead measuring the difference between them based on their ability to do the job they are being trained towards. If the approximate activation functions proposed here were to be adopted they would need to be consistent between their own implementations.

It is currently not possible to analytically determine the best choice of activation function for a specific network architecture. Similarly it is not possible to prove that a particular approximation is a better choice in all cases. Instead I seek to empirically justify my choices through end-to-end measurements on three popular representative machine learning tasks covering three popular network architectures.

MNIST Classifier I consider the task of classifying images in the MNIST dataset and use a network comprising convolutional and dense layers inspired by the design of successful neural network structures [146] based on the implementation from a selection of provided models for Flux [83]. While ReLU is sometimes used for convolutional image classification tasks for it's speed and simplicity, `tanh` and `sigm` have been used more commonly in the past [91, 98, 95] and have some desirable properties which can reduce training times in some scenarios [31].

MNIST Autoencoder Autoencoders are common in the area of generative machine learning and are often evaluated using the MNIST dataset [99]. I implemented a simple autoencoder to work with this dataset. Autoencoders are compatible with many different activations when structured with different configurations [155, 23, 122, 162, 157, 28]. The model used is the simplest example of an autoencoder and as such is compatible with `ReLU`, `sigm` and `tanh` activations.

CharRNN Sequence-to-sequence problems are another common task. A common LSTM network layout with 2 hidden LSTM layers was selected for text generation that takes a text dataset and learns to generate similar text. LSTM cells make use of both `sigm` and `tanh` activation functions and so approximations for both are considered.

Each approximation was used in turn to train three networks for a fixed number of epochs, recording the loss and the total time taken. These values were then compared to the non-approximated activation functions to compute the relative loss and relative time taken (Table 7.1).

[3] *Loss* in neural network training is the difference in the output of the network from the correct answer.

				Loss		Time (s)		
		Activation		Abs.	Rel.	Abs.	Rel.	Choice
Convnet (MNIST)		ReLU		25.99	1.000	486.9	1.000	
	Sigmoid	sigm		20.23	1.000	889.5	1.000	
		sigm_fastexp_2		17.21	0.851	643.5	0.723	**Ranged**
		sigm_fastexp_512		20.39	1.008	798.5	0.898	**Safe**
		sigm_taylor_9		∞	∞	-	-	
		sigm_pade_4_4		21.00	1.038	702.9	0.790	
		ultra_fast_sigmoid†		20.63	1.020	743.2	0.836	
		word2vec†		456.1	22.55	833.9	0.937	
	Tanh	tanh		16.64	1.000	1126	1.000	
		tanh_cont_4		16.54	0.994	654.0	0.581	
		tanh_taylor_9		∞	∞	-	-	
		tanh_pade_4_4		13.98	0.840	712.8	0.633	**Safe**
		serp		14.93	0.897	523.7	0.465	**Ranged**
		serp_clamp		18.89	1.135	604.8	0.537	
Autoencoder (MNIST)		ReLU		1.441	1.000	25.87	1.000	-
	Sigmoid	sigm		4.166	1.000	35.54	1.000	
		sigm_fastexp_2		3.924	1.006	28.33	0.797	**Ranged**
		sigm_fastexp_512		4.167	1.000	31.47	0.886	**Safe**
		sigm_taylor_9		6.788	1.630	32.74	0.921	
		sigm_pade_4_4		4.161	0.999	30.56	0.860	
		ultra_fast_sigmoid†		4.210	1.011	32.48	0.914	
		word2vec†		13.94	3.347	32.58	0.917	
	Tanh	tanh		2.234	1.000	37.14	1.000	
		tanh_cont_4		2.256	1.010	30.39	0.818	
		tanh_taylor_9		2.237	1.001	35.73	0.962	
		tanh_pade_4_4		2.242	1.004	32.48	0.875	**Safe**
		serp		2.147	0.961	28.36	0.770	**Ranged**
		serp_clamp		2.151	0.963	31.36	0.845	
CharRNN	**LSTM**	ReLU	ReLU	NaN	-	-	-	
		sigm	tanh	79.16	1.000	1502.93	1.000	
		sigm_fastexp_2	tanh	82.30	1.040	1406.603	0.936	
		sigm_fastexp_512	tanh	77.65	0.981	1401.893	0.933	
		sigm_taylor_9	tanh	∞	∞	-	-	
		sigm_pade_4_4	tanh	78.01	0.985	1462.484	0.973	
		sigm	tanh_cont_4	78.68	0.994	1361.133	0.906	
		sigm	tanh_taylor_9	∞	∞	-	-	
		sigm	tanh_pade_4_4	77.99	0.985	1407.648	0.937	
		sigm	serp	78.29	0.989	1303.92	0.868	
		sigm	serp_clamp	78.46	0.991	1367.542	0.910	
		sigm_fastexp_2	**serp**	79.82	1.008	1332.127	0.886	**Ranged**
		sigm_fastexp_2	serp_clamp	82.81	1.046	1184.179	0.788	
		sigm_fastexp_512	tanh_cont_4	∞	∞	-	-	
		sigm_fastexp_512	tanh_pade_4_4	79.34	1.002	1446.656	0.963	
		sigm_fastexp_512	serp	77.76	0.982	1150.14	0.765	
		sigm_fastexp_512	**serp_clamp**	79.24	1.001	1155.745	0.769	**Safe**
		ultra_fast_sigmoid†	tanh	78.88	0.996	1450.332	0.965	
		word2vec†	tanh	100.7	1.271	1561.443	1.039	

Table 7.1: Performance results for the range of approximations considered. The Rel. columns indicate performance relative to the replaced function (smaller values are better).
(†) We discuss the performance of `ultra_fast_sigmoid` and `word2vec` later.

Relative values are with respect to the function being replaced (rather than ReLU) and smaller relative values indicate better performance. For the MNIST workloads an Intel Xeon E5-2673 v3 @ 2.40GHz (14GB RAM) was used and a dual-core Intel Xeon E5-2673 v4 @ 2.30GHz (8GB RAM) was used for the RNN workload. The benchmarks ran on Azure cloud machines and virtual machine images for Azure are provided here[4] to allow easy replication of the results.

In most cases the replacement activation functions either converged with similar loss to the original function or failed to converge entirely (marked ∞ in the table). A few instances were found (such as sigm_fastexp_2 in ConvNet) for which the loss was drastically lower (15%). This case is another example of the difficulty of making definitive statements about the correct choice of activation function. In all cases where the training loss converged (except for my LSTM Word2Vec), the approximations resulted in a reduced overall training time.

It was found that two functions (sigm_fastexp_512 and tanh_pade_4_4) produced the best reduction in training time (10%–37%) whilst working in all cases.These are marked as the chosen *safe* approximations. Functions (such as sigm_fastexp_2 and serp) were also found which produced even better reduction in training times (20%–53%) but have such significant divergence in form from the target functions that it should not be argued that they are a suitable replacement in all cases. These are marked as *ranged* approximations suitable for networks (such as these) where the activation input values are roughly within the range of -5 to 5.

Performance improvements were also found when using approximations for inference rather than training. For the character-based RNN (LSTM) model the safe approximations offered ~8% savings where as the ranged approximations allowed for savings of up to 20% when performing 1000 sequential inferences. This is potentially more significant than the reduction in training times since inference is often performed on restricted hardware such as mobile devices.

7.4 Comparison with ultra_fast_sigmoid

ultra_fast_sigmoid is a fast sigm implementation in popular machine learning framework library Theano [13]. It is implemented as a piece-wise approximation. It is one of only a few approximations available openly as standard in popular machine learning libraries.

I compared ultra_fast_sigmoid to sigm_fastexp_512. Figure 7.3 shows the much greater approximation error in ultra_fast_sigmoid. For the Autoencoder and CharRNN workloads sigm_fastexp_512 results in lower loss in less time. For the Convnet workload sigm_fastexp_512 is slightly slower but with drastically lower loss.

This makes ultra_fast_sigmoid appear to be a less appealing approximation.

[4] https://drive.google.com/file/d/1trqpemv9BScwt88Xd69zpZKGWM2RMXs3/view?usp=sharing

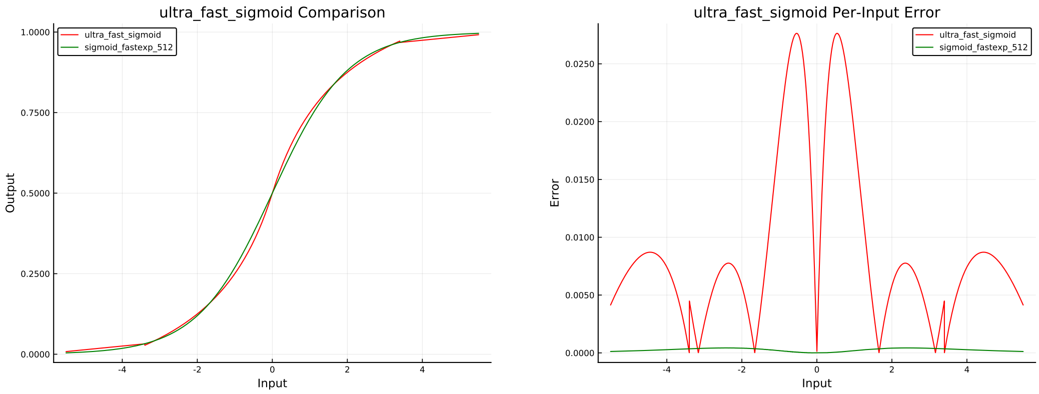


Figure 7.3: Shape and relative error of Theano's `ultra_fast_sigmoid` compared to `sigm_fastexp_512`.

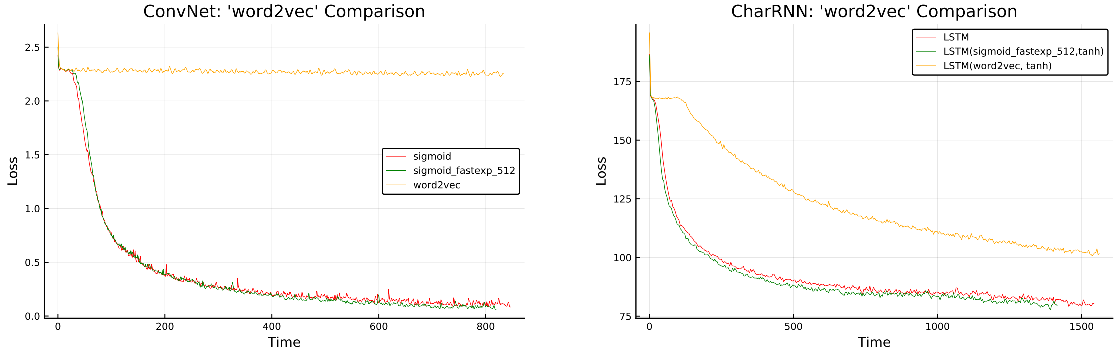


Figure 7.4: Loss over time for the `word2vec` table based sigmoid function.

7.5 Comparison with Word2Vec Lookup Table

Word2Vec makes use of shallow 2-layer networks to create word embeddings. In these models the authors used an approximated sigmoid implementation which makes use of a 1000 element lookup table. I have compared this lookup table to my approximations in my test models.

My results (Table 7.1) show that applying this approximation results in more loss for a similar or slightly reduced training time. When looking at the loss over time, Figure 7.4 shows that in some cases training fails to make progress. I believe this occurs due to the lack of interpolation in the table lookup resulting in quantisation of outputs. As a result small incremental changes to the weight values may not result in a change of output. This could potentially be mitigated with use of a different optimiser or by adding interpolation (at the cost of a performance reduction).

Again, `sigmoid_fastexp_512` results in lower loss in less training time.

7.6 Approximations in TensorFlow

TensorFlow is one of the most commonly used machine learning libraries. Despite the fact that it is a Python library it achieves high performance through optimised native code implementation of the core functions. As such this makes experimentation with novel activation functions difficult:

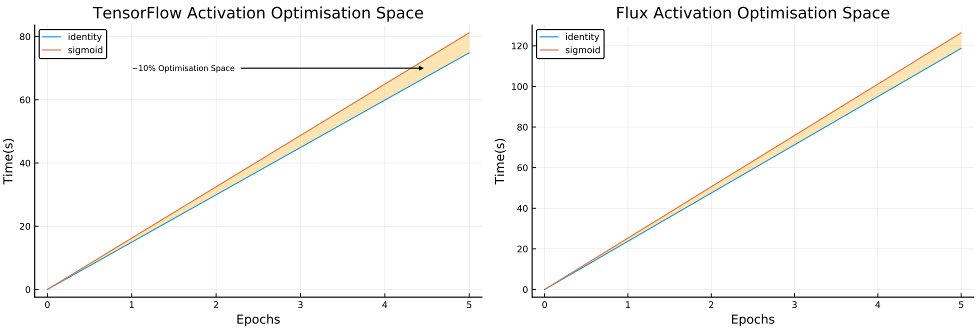


Figure 7.5: Training time per epoch for the MNIST classifier in TensorFlow and Flux using `sigm` and `identity` activation functions.

one must inject a low-level implementation and then provide a mechanism to reference it from the high-level code. Flux does not suffer from this issue because the entire system is implemented in Julia (a relatively high performance language) and alternative activation functions can be straightforwardly applied on a level playing field with the default options.

Despite being unable to directly evaluate my new activation functions in TensorFlow I was able to identify the optimisation space by measuring the performance of the simplest possible activation function, the identity function.

Figure 7.5 shows the time taken to train a simple MNIST classifier (one convolution layer and two dense layers) when using `sigm` and `identity` in both TensorFlow (left) and Flux (right). The approximations discussed in this paper fall within the region between these two lines. Even with this simple model this demonstrates that there is scope for improving model performance in TensorFlow by optimising activation functions.

7.7 Threats to Validity

Although I have tried to evaluate my activation functions on three representative workloads it is not possible to say for sure how well they will work in the general case. However, the relative errors in my safe approximations are so small that I would expect them to be drop-in replacements.

Neural networks are commonly trained offline on large compute clusters whereas inference happens with interactive latencies and increasingly on limited hardware (such as mobile phone handsets). The majority of my results focus on training times because training loss provides a convenient measurement to check that the activation function is performing well. However, I also found that my approximations improve inference and mention this in Section 7.3.

Our results only consider the performance on CPUs whereas much training (and inference) happens on GPUs or specialist hardware such as TPUs. I therefore cannot say how the approximations perform in these circumstances. I argue that the safe approximations are useful even if only applied to CPUs since they generally reduce training times with no impact on loss. It would be particularly interesting to consider hardware implementations of these approximations for specialised hardware. This is left for future work.

7.8 Conclusion

I have shown that approximation of activation functions in neural networks can improve the training and inference time without a negative effect on the accuracy of the network.

I investigated a range of functions and propose two safe approximations `sigm_fastexp_512` and `tanh_pade_4_4` for `sigm` and `tanh` respectively. These approximations produce faster training and inference times without damaging prediction accuracy. The ranged approximations produced even larger speedups but will not work for all networks.

As such I think these functions are candidates for inclusion as standard options in machine learning libraries. I also showed that these functions outperform existing approximations in these libraries.

In the future I hope to expand on this work by integrating approximation as a standard option into many machine learning libraries so that it can be used to improve training time on a larger scale. Additionally I wish to analyse the effect of approximations on large and complex networks and hardware to understand if there is a structure which may cause approximations to not be beneficial.

This chapter has shown that approximations of the elementary functions, and the functions which are derived from them, can provide a tangible benefit without harming the output of the application. Meaning, that approximations to the appropriate level of error can be a valid optimisation step.

Chapter 8

Conclusion

In this dissertation, I set out to show that approximation is being used in existing mathematical functions, and therefore that some tolerance to error was already being accepted by the mathematical development community. I theorised that, if the current levels of error are acceptable, there may be a larger optimisation space to explore if we allow for further error or even if we further optimised towards existing error tolerances.

I have measured the unreported error in existing mathematical software and used that information to explore and show the effect those errors are having on existing applications when bit-perfect reproducibility is required. These errors were shown in places where the average user may not notice the problems.

To solve the problems where the inaccuracy was causing unacceptable levels of error, such as where switching mathematical library results in a different neural network output, I proposed a new approach to calculating correctly rounded 32-bit results for the elementary functions, as required by IEEE-754. I also demonstrated how the performance of these new correctly rounded functions could be improved by using coarser approximations at a higher precision than the target output.

As an alternative to correct-rounding when dealing with programs that are error-tolerant, I presented tools and techniques to reduce the accuracy of existing implementations to a tolerable limit to gain a performance improvement. This showed that the existing mathematical library implementations which are already incorrect were not fully exploiting the worst-case error to approach the maximum performance available.

I then applied less accurate but better-performing elementary functions to the field of machine learning by replacing the activation functions in neural networks with fast approximations. In doing so, I showed that neural networks are not reliant on the activation functions having correctly rounded results and it is possible to significantly decrease training times by replacing the activation functions with cheaper alternatives to reduce the cost of the training process without unacceptably affecting the accuracy of the final trained network.

An important highlight of this work was our discovery of the choice to not follow the IEEE-754 standard by many of the standard mathematical libraries, which not obvious to a regular user and, for many libraries, not documented. Our highlighting of this behaviour shines a light on possible silent portability and unexpected errors which may be untested in the wild. The proposed correctly rounded solution for 32-bit mathematical functions is available to solve this problem.

I am also keen to bring attention to large performance improvements I was able to show in neural network training time by identifying that the training process was error-tolerant and therefore did not need to spend as much compute power on the activation functions which were being used. I want to encourage developers who train networks for hundreds of hours to adopt faster alternatives for activation functions so that less time is needed for training. Other mathematical software which is error-tolerant will also benefit from similar investigations.

Our findings have shown that there is a lot of industry work required for the existing standard mathematical solutions to meet the levels of quality desired by the standards they are aiming to follow. Our solutions for one aspect of this problem may help contribute to that, but due to the pervasiveness of Hyrum's Law[1], many existing solutions inevitably rely either on the current performance or current inaccuracy in some form, making changes in this domain non-trivial and potentially breaking some software in what might be subtle or hard to identify ways.

One example of the reliance on existing answers was when I accidentally found that popular compilers are not returning correctly rounded results when performing performing constant folding to write results from the system maths library. I want to implement changes to popular compilers and the C++ standard to prevent this optimisation. We would prefer it was moved under the `-ffast-math` flag so that when a developer is using this optimisation they are aware of the mathematical errors it may incur.

8.1 Future Work

Our findings during the exploration of approximation for mathematical functions have shown that the area is currently lacking a lot of measurement and tracking of what is actually being used in real software. I have presented techniques to show how approximation can be used to get better performance while still within acceptable error boundaries, such as our exploration of a one ULP max-error implementation of `sin`, but have only applied to this one function in this dissertation. I hope that further work is applied to automating the process that was used in that example so that it can be easily applied to all the elementary functions at arbitrary levels of error. This would provide a means to track the existing behaviours of elementary mathematical functions and open the door to research into optimisation design-space exploration and further

[1] "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody." [166]

ease the adoption of safe approximation into real-world computing by providing information for pre-made, bounded, safe approximations.

I also applied our approximations to the development of neural network models. Neural networks are a rapidly growing area of research with many researchers exploring different numerical representations to minimise the cost of training and using these networks. Our research shows an avenue of research which has not been investigated in great detail before. Those looking to gain increased performance in more complex networks than I was able to explore may find benefit in researching how my, or other, approximations of the activation functions can improve the performance of their networks without penalty to the final result.

The area I see as having the biggest potential for further work is in the standardisation of the results of the elementary functions. A significant portion of this work was spent exploring the existing error in functions which should not be prone to error. There is a lot of work needed to bring all the commonly used mathematical functions together to return the same results and conform with existing standards, or alternatively modifying the existing standards to allow for error. Whichever way the decision is made to bring determinism and portability to mathematical computing, there is a lot of work to be done.

I want to note the difference between correctly rounded results and deterministic results. The former will guarantee the same results for all implementation and those results being correct for the given function and input. But, simply having deterministic results still allows for approximations to one or more ULP, as long as all implementations produce the same output for the same input. With the latter, code still remains safely portable without forcing the higher computational cost of correct-rounding. Choosing a *canonical* set of results with a ULP error of one or more for each function is more complicated and would be something that requires further work to choose what those results should be.

I have attempted to present as detailed discussion and evidence for our approaches as possible, but as with all things related to performance and optimisation — there are always more machines and configuration to specialise for.

8.2 Final Remarks

I hope that this dissertation has been able to present not only the current state of approximation in mathematical functions, but also to have provided interesting insight into how to develop and apply safe approximations to gain the performance benefit they can bring to error tolerant programs.

It is my belief that by providing the tools, knowledge and background for new approximation approaches I can improve the field of approximate computing in a small way, and help to bring the use of these techniques into the mainstream as part of the normal development process. I see this as one way in which mathematical computing can become more efficient, better utilise available resources, and help computing achieve better results.

Bibliography

[1] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008.

[3] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019*, 2019.

[4] glibc: Known maximum errors in math functions. `https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html`, 2020.

[5] 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.

[6] Mohammed Ba Alawi. *What Every Programmer Should Know about Floating Point Arithmetic*. PhD thesis, University of Leeds, School of Computing, 2004.

[7] Jonas S Almeida, Joao A Carrico, Antonio Maretzek, Peter A Noble, and Madilyn Fletcher. Analysis of genomic sequences by chaos game representation. *Bioinformatics*, 17(5): 429–437, 2001.

[8] Todd Arbogast. Disasters due to rounding error. `https://web.ma.utexas.edu/users/arbogast/misc/disasters.html`.

[9] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[10] Michael F Barnsley. Fractal modeling of real world images. In *The science of fractal images*, pages 219–242. Springer, 1988.

[11] Michael F Barnsley and Andrew Vince. The chaos game on a general iterated function system. *Ergodic theory and dynamical systems*, 31(4):1073–1079, 2011.

[12] Louis Bavoil and Miguel Sainz. Screen space ambient occlusion. *NVIDIA developer information: http://developers.nvidia.com*, 6, 2008.

[13] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[14] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[15] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[16] DRB Bish and JRA Cooper. Guide to the NPL algorithms library. 1976.

[17] Cheryl M Black, , and Robert Burton Thomas H. Miller. The need for an industry standard of accuracy for elementary-function programs. *ACM Transactions on Mathematical Software (TOMS)*, 10(4):361–366, 1984.

[18] James F Blinn. Floating-point tricks. *IEEE Computer Graphics and Applications*, 17(4): 80–84, 1997.

[19] Andrea Bocco, Yves Durand, and Florent de Dinechin. Hardware support for unum floating point arithmetic. In *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, pages 93–96, 2017. doi: 10.1109/PRIME.2017.7974115.

[20] Ronald F Boisvert, José Moreira, Michael Philippsen, and Roldan Pozo. Java and numerical computing. *Computing in Science & Engineering*, 3(2):18–24, 2001.

[21] David Braben. Rules can be beautiful. *TEDxAlbertopolis*, 2012. https://www.ted.com/tedx/events/7663.

[22] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[23] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*, 2015.

[24] Simon Byrne. Remez.jl source code. `https://github.com/simonbyrne/Remez.jl`.

[25] Florian Cajori. Horner's method of approximation anticipated by Ruffini. *Bulletin of the American Mathematical Society*, 17(8):409–414, 1911.

[26] Mark Carney. Signed digital documents, U.S. Patent 20100037062A1, 2010-02-11. URL `https://patents.google.com/patent/US20100037062A1/en`.

[27] Luis Ceze and Adrian Sampson. Approximate computing: Unlocking efficiency with hardware-software co-design. *GetMobile: Mobile Computing and Communications*, 20 (3):12–16, 2017.

[28] Minmin Chen, Zhixiang Xu, Kilian Weinberger, and Fei Sha. Marginalized denoising autoencoders for domain adaptation. *arXiv preprint arXiv:1206.4683*, 2012.

[29] Vinay K Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the 47th Design Automation Conference*, pages 555–560. ACM, 2010.

[30] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

[31] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[32] William J Cody and William M Waite. *Software manual for the elementary functions*. Prentice-Hall, 1980.

[33] Pierluigi Crescenzi and Viggo Kann. Approximation on the web: A compendium of NP optimization problems. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 111–118. Springer, 1997.

[34] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[35] Catherine Daramy-Loirat, David Defour, Florent De Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, and Jean-Michel Muller. CR-LIBM A library of correctly rounded elementary functions in double-precision. 2010.

[36] Marc Daumas, Christophe Mazenc, Xavier Merrheim, and Jean-Michel Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. In *J. UCS The Journal of Universal Computer Science*, pages 162–175. Springer, 1996.

[37] Florent de Dinechin, Alexey V Ershov, and Nicolas Cast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 288–295. IEEE, 2005.

[38] Theodorus Jozef Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[39] Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[40] Premkumar Devanbu, PW-L Fong, and Stuart G Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 126–135. IEEE, 1998.

[41] Wilco Dijkstra. glibc commit: Speedup tanf range reduction. `https://github.com/bminor/glibc/commit/900fb446eb8172c54cdaed85107bc783ee50673a#diff-bcee6b06b04074527741a2d9241c293ebb2516e4aa9bb1ea51258dd39c77aa61`, 2018.

[42] Wilco Dijkstra. glibc code: tanf new range reduction. `https://github.com/bminor/glibc/blob/master/sysdeps/ieee754/flt-32/s_tanf.c#L30`, 2018.

[43] Wilco Dijkstra. glibc code: old range reduction. `https://github.com/bminor/glibc/blob/60bcac09c0be46f56583e260f5fef849a5845074/sysdeps/ieee754/flt-32/e_rem_pio2f.c#L90`, 2018.

[44] CUDA Toolkit Documentation. Section e "programming guide" v10.1.168. `https://docs.nvidia.com/cuda/archive/10.1/`, 2019.

[45] Junyu Dong, Jun Liu, Kang Yao, Mike Chantler, Lin Qi, Hui Yu, and Muwei Jian. Survey of procedural methods for two-dimensional texture generation. *Sensors*, 20(4):1135, 2020.

[46] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[47] Iain S Duff. Some notes on numerical library construction. Technical report, CM-P00068530, 1979.

[48] Hadi Esmaeilzadeh, Kangqi Ni, and Mayur Naik. Expectation-oriented framework for automating approximate programming. Technical report, Georgia Institute of Technology, 2013.

[49] Leonhard Euler. *Introductio in analysin infinitorum*, volume 2. Apud Marcum-Michaelem Bousquet & Socios, 1748.

[50] Warren Ferguson, Marius Cornea, Cristina Anderson, and Eric Schneider. The difference between x87 instructions fsin, fcos, fsincos, and fptan and mathematical functions sin, cos, sincos, and tan. *Intel Corporation. Available at:* `https://software.intel.com/en-us/articles/thedifference-between-x87-instructions-and-mathematical-functions`, 2015.

[51] Brian Ford. Developing a numerical algorithms library. *IMA Bulletin*, 8:332–336, 1972.

[52] Brian Ford and DK Sayers. Developing a single numerical algorithms library for different machine ranges. *ACM Transactions on Mathematical Software (TOMS)*, 2(2):115–131, 1976.

[53] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13–es, 2007.

[54] David Fowler and Eleanor Robson. Square root approximations in old babylonian mathematics: Ybc 7289 in context. *Historia Mathematica*, 25(4):366–378, 1998.

[55] Phyllis Fox, Andrew P Hall, and Norman L Schryer. The PORT mathematical subroutine library. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):104–126, 1978.

[56] FreeBSD. Freebsd source. `https://cgit.freebsd.org/`, 2021.

[57] Rūsiņš Freivalds. Fast probabilistic algorithms. In *International Symposium on Mathematical Foundations of Computer Science*, pages 57–69. Springer, 1979.

[58] L Wayne Fullerton. Portable special function routines. In *Portability of Numerical Software*, pages 452–483. Springer, 1977.

[59] Shmuel Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate scientific computations*, pages 1–16. Springer, 1986.

[60] Shmuel Gal. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software (TOMS)*, 17(1):26–45, 1991.

[61] Deepak Ganesan, Deborah Estrin, and John Heidemann. Dimensions: Why do we need a new data handling architecture for sensor networks? *ACM SIGCOMM Computer Communication Review*, 33(1):143–148, 2003.

[62] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. 1999.

[63] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995.

[64] Brian Gladman, W Hart, J Moxham, et al. MPIR - multiple precision integers and rationals, 2010.

[65] GNU. GNU Libc. `https://www.gnu.org/software/libc/`, 2021.

[66] Matt Godbolt. Compiler Explorer. *`www.godbolt.org`*.

[67] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[68] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[69] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[70] Google. Fdlibm package in android. `https://android.googlesource.com/platform/external/fdlibm/`, 2021.

[71] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley Professional, 2000.

[72] Torbjörn Granlund. GNU multiple precision arithmetic library. *http://gmplib. org/*, 2010.

[73] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[74] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional LSTM. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 273–278. IEEE, 2013.

[75] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2016.

[76] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2013.

[77] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, volume 47, pages 123–134. ACM, 2012.

[78] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro*, 33(3):58–66, 2013.

[79] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1474–1479. IEEE, 2017.

[80] Cecil Hastings. *Approximations for Digital Computers.* Princeton University Press, USA, 1955. ISBN 0691079145.

[81] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019.

[82] Randall Hyde. *Write Great Code, volume 2: Thinking low-level, writing high-level*, volume 2. No Starch Press, 2020.

[83] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[84] ISO/IEC. ISO/IEC 9899:2018: Programming languages — C. 2018.

[85] ISO/IEC. ISO/IEC 14882:2020: Programming languages — C++. 2020.

[86] Andreas Jaeger. OpenLibm, 2016. https://openlibm.org/.

[87] Huw Jones. Dürer, Gaskets and Barnsley's chaos game. In *Computer graphics forum*, volume 9, pages 327–332. Wiley Online Library, 1990.

[88] ISO Jtc. Sc22/wg14. iso/iec 9899: 2011. *Information Technology-Programming Languages-C*, 2011.

[89] William Kahan and John Palmer. On a proposed floating-point standard. *ACM SIGNum Newsletter*, 14(si-2):13–21, 1979.

[90] William Kahan, Joseph D Darcy, Elect Eng, and High-Performance Network Computing. How Java's floating-point hurts everyone everywhere. In *ACM 1998 Workshop on Java for high-performance network computing*, page 81. Stanford University, 1998.

[91] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[92] Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. Clumsy value cache: An approximate memoization technique for mobile GPU fragment shaders. In *Workshop on Approximate Computing (WAPCO'15)*, 2015.

[93] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78, 2008.

[94] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68697-2.

[95] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[96] Sunil Kumar, Liyang Xu, Mrinal K Mandal, and Sethuraman Panchanathan. Error resiliency schemes in H. 264/AVC standard. *Journal of Visual Communication and Image Representation*, 17(2):425–450, 2006.

[97] Olga Kupriianova and Christoph Lauter. Metalibm: A mathematical functions code generator. In *International Congress on Mathematical Software*, pages 713–717. Springer, 2014.

[98] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1): 98–113, 1997.

[99] Yann LeCun, Corinna Cortes, and Christopher Burges. MNIST dataset. `http://yann.lecun.com/exdb/mnist`, 1998.

[100] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A Jacobson, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1560–1565. IEEE, 2010.

[101] Vincent Lefèvre, J-M Muller, and Arnaud Tisserand. Toward correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, 1998.

[102] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. The Table Maker's Dilemma. 1998. [Research Report] LIP RR-1998-12, Laboratoire de l'informatique du parallélisme. 1998, 2+16p. ffhal-02101765f.

[103] LLVM. LLVM's analysis and transform passes, 2018. URL `https://llvm.org/docs/Passes.html`.

[104] Chris Lomont. Fast inverse square root. *Technical Report*, 32, 2003.

[105] Timothy Lottes. FXAA. *NVIDIA white paper*, 2:8, 2011.

[106] Lu Luo. Software testing techniques. *Institute for Software Research International, Carnegie Mellon University, Pittsburgh, PA*, 15232(1-19):19, 2001.

[107] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGPLAN Notices*, volume 22, pages 122–126. IEEE Computer Society Press, 1987.

[108] Morgan McGuire and Michael Mara. Efficient GPU screen-space ray tracing. *Journal of Computer Graphics Techniques (JCGT)*, 3(4):73–85, 2014.

[109] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[110] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 127–139. IEEE Computer Society, 2014.

[111] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *arXiv preprint arXiv:1310.4546*, 2013.

[112] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, volume 49, pages 309–328. ACM, 2014.

[113] Asit K Mishra, Rajkishore Barik, and Somnath Paul. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[114] Michal Misiurewicz. Strange attractors for the Lozi mappings. *Annals of the New York Academy of Sciences*, 357(1):348–358, 1980.

[115] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):62, 2016.

[116] Bert Moons, Bert De Brabandere, Luc Van Gool, and Marian Verhelst. Energy-efficient convnets through approximate computing. In *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*, pages 1–8. IEEE, 2016.

[117] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 29–40. IEEE, 2003.

[118] Jean-Michel Muller. *Elementary functions*. Springer, 2006.

[119] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*, volume 1. Springer, 2018.

[120] David Nassimi and Sartaj Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, 100(2):101–107, 1981.

[121] Rafi Nave. Implementation of transcendental functions on a numerics processor. *Microprocessing and Microprogramming*, 11(3-4):221–225, 1983.

[122] Andrew Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[123] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[124] John F Palmer. The Intel® 8087 numeric data processor. In *Proceedings of the National Computer Conference*, pages 887–893, 1980.

[125] Matt Parker. *Humble Pi: A Comedy of Maths Errors*. Penguin UK, 2019.

[126] Kenneth Perlin. Standard for Perlin noise, 2005. US Patent 6,867,776.

[127] Eric Quinnell, Earl E Swartzlander, and Carl Lemonds. Floating-point fused multiply-add architectures. In *2007 Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers*, pages 331–337. IEEE, 2007.

[128] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

[129] Eugene Y Remez. Sur la détermination des polynômes d'approximation de degré donnée. *Comm. Soc. Math. Kharkov*, 10(196):41–63, 1934.

[130] Jarrett Revels. Benchmarktools.jl. `https://github.com/JuliaCI/BenchmarkTools.jl`, 2021.

[131] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[132] Martin Rinard. Obtaining and reasoning about good enough software. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 930–935. IEEE, 2012.

[133] Martin C Rinard. Living in the comfort zone. *ACM SIGPLAN Notices*, 42(10):611–622, 2007.

[134] MRD Rodrigues, JHP Zurawski, and JB Gosling. Hardware evaluation of mathematical functions. *IEEE Proceedings (Computers and Digital Techniques)*, 128(4):155–164, 1981.

[135] Edward J. Rosten and Oliver J. Rosten. More constexpr for <cmath> and <complex>. (P1383R0), 2019.

[136] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. ASAC: Automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN Notices*, volume 49, pages 95–104. ACM, 2014.

[137] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 13–24. IEEE, 2013.

[138] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. *ACM SIGPLAN Notices*, 49 (4):35–50, 2014.

[139] Adrian Sampson. The case for compulsory approximation. In *Workshop on Approximate Computing Across the Stack*, 2016.

[140] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[141] Adrian Sampson, Luis Ceze, and Dan Grossman. Good-enough computing. *IEEE Spectrum*, 50(10):54–59, 2013.

[142] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

[143] Michael Schulte. Floating Point Arithmetic in AMD Processors, 2015. URL http://arith22.gforge.inria.fr/slides/s1-schulte.pdf.

[144] Charles Severance. IEEE 754: an interview with William Kahan. *Computer*, 31(3): 114–115, 1998.

[145] Arthur Shek, Dylan Lacewell, Andrew Selle, Daniel Teece, and Tom Thompson. Art-directing Disney's Tangled procedural trees. In *ACM SIGGRAPH 2010 Talks*. 2010.

[146] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[147] Alex J Smola and Bernhard Schölkopf. Sparse greedy matrix approximation for machine learning. 2000.

[148] Phillip Stanley-Marbell, Martin Rinard, et al. Error-efficient computing systems. *Foundations and Trends® in Electronic Design Automation*, 11(4):362–461, 2017.

[149] David G Steer and SR Penstone. Digital hardware for sine-cosine function. *IEEE Computer Architecture Letters*, 26(12):1283–1286, 1977.

[150] Damien Stehlé and Paul Zimmermann. Gal's accurate tables method revisited. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 257–264. IEEE, 2005.

[151] Sun Microsystems. FDLIBM. *https://www.netlib.org/fdlibm/*, 1993.

[152] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[153] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[154] Gabor Szeg. *Orthogonal polynomials*, volume 23. American Mathematical Soc., 1939.

[155] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[156] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. Technical report, Argonne National Lab., IL (USA), 1991.

[157] George Toderici, Sean M O'Malley, Sung Jin Hwang, Damien Vincent, David Minnen, Shumeet Baluja, Michele Covell, and Rahul Sukthankar. Variable rate image compression with recurrent neural networks. *arXiv preprint arXiv:1511.06085*, 2015.

[158] Anne Sjerp Troelstra. Non-extensional equality. *Fundamenta Mathematicae*, 82(4): 307–322, 1975.

[159] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016.

[160] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: energy-efficient neuromorphic systems using approximate computing. In *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*, pages 27–32. IEEE, 2014.

[161] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Annual Design Automation Conference*, page 120. ACM, 2015.

[162] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[163] Xiaojun Wang and Miriam Leeser. Vfloat: A variable precision fixed-and floating-point library for reconfigurable hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(3):1–34, 2010.

[164] Eric W Weisstein. Chebyshev polynomial of the first kind. `https://mathworld.wolfram.com/`, 2003.

[165] James H Wilkinson. Error analysis of floating-point computation. *Numerische Mathematik*, 2(1):319–340, 1960.

[166] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020.

[167] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 5. IEEE Computer Society, 2007.

[168] Hang Zhang, Mateja Putic, and John Lach. Low power GPGPU computation with imprecise hardware. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[169] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. ApproxANN: an approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 701–706. EDA Consortium, 2015.

[170] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, volume 47, pages 441–454. ACM, 2012.

[171] Paul Zimmermann. Why transcendentals and arbitrary precision? *IEEE 754 Revision Committee*, 2005.

[172] Paul Zimmermann. Reliable computing with GNU MPFR. In *International Congress on Mathematical Software*, pages 42–45. Springer, 2010.

[173] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software (TOMS)*, 17(3):410–423, 1991.

Appendix A

Standard Mathematical Definitions

We adopt standard definitions for common mathematical terms which we include below for completeness.

A.1 Mathematical Approximation Techniques

A.1.1 Table Lookup

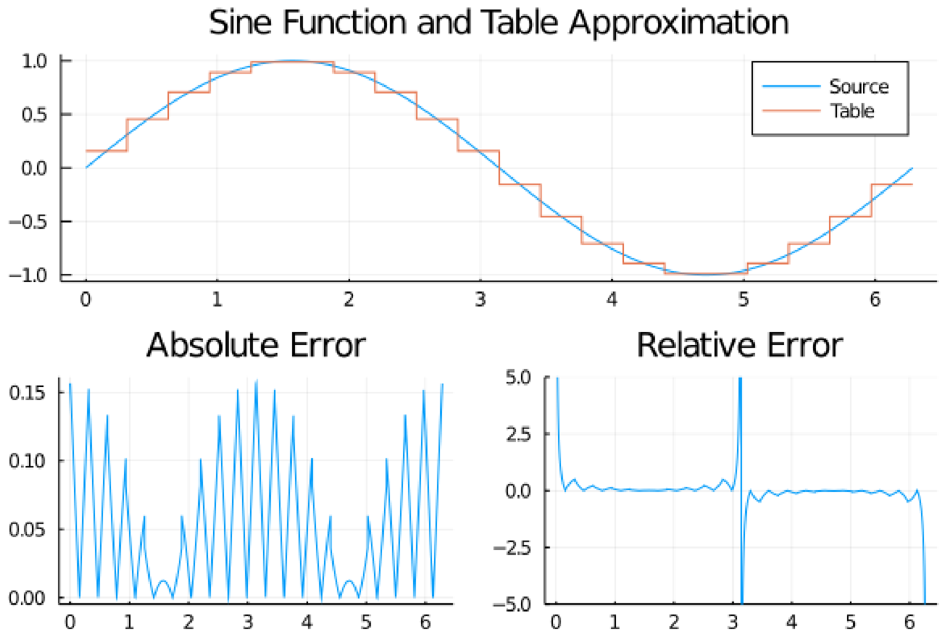
This section gives a step-by-step example of how to apply techniques for function approximation via table lookup. Ranging from direct to gradient assisted interpolated lookup.

A.1.1.1 Direct Lookup

Direct lookup is a table based approximation approach where samples of the target function are stored in a table and can be directly queried.

The benefits of direct lookup are the guarantee of correct answers for any result stored in the table. In the past, this was once the fastest, and sometimes the only viable way to calculate some elementary functions. However, with the increase in processing power of computers, the relative lag in memory performance, and the introduction of more complex memory architectures, look-up performance for this style of implementation can have mixed results. Very fast for sequential reads, but very slow for random access. Mileage may vary based on your cache behaviour! If we wanted to guarantee every answer for a specific function, this is the safest way and no longer prohibitively expensive to store for all 32-bit inputs. Storing all outputs for a 32-bit function takes approximately 16GB of RAM - – less than is commonly installed on modern high-end workstations.

A simple demonstration of direct table lookup with a limited table size of twenty elements can be seen in Figure A.1 where a table of twenty input-output pairs are stored to approximate the `sin` function. The highest absolute error can be found at the point on the curve where the



LOOKUP TABLE (n=20)	
`sin` *Input*	*Output*
0.0	0.0
0.3141...	0.3090...
0.6283...	0.5877...
0.9424...	0.8090...
1.2566...	0.9510...
⋮	⋮

$$k = floor(x/step)$$
$$y = table[k]$$

Figure A.1: An example of a small table used for direct lookup of n=20 for entries the Sine function.

gradient is maximum as this introduces the largest difference in output value between the two straddling values which are stored.

Direct-table lookup has some problems, as large areas of the curve are significantly mismatched and the differences in error between results are highly varied.

A.1.1.2 Interpolated Lookup

To improve on the direct table lookup reads, we can use interpolation at the cost of an additional table read of the previous or next value. For this approach, the enclosing values from the table for the input being queried are read and the value returned is an interpolation of these two values.

This approach is linear and therefore the maximum error shifts from the areas with max gradient, to the areas with the highest change in gradient (the highest 2nd derivative). This is shown in Figure A.2. The figure also shows that both absolute and relative error is greatly reduced, as would be expected.

A.1.1.3 Gradient Tables Lookup

With interpolated tables, the fit to the source is best where the gradient at the two points being interpolated is similar. When it is not, such as at the peaks and troughs of the Sine function, there is a significant discrepancy between the true value and the approximation.

At the cost of storing the gradient at each sample position and the cost of another two gradient look-ups at run-time, it is possible to improve the accuracy at the points where the gradient changes rapidly. The stored gradients can be interpolated for the current position between the two samples and used to bias the base interpolated position with the weight of the gradient in that direction.

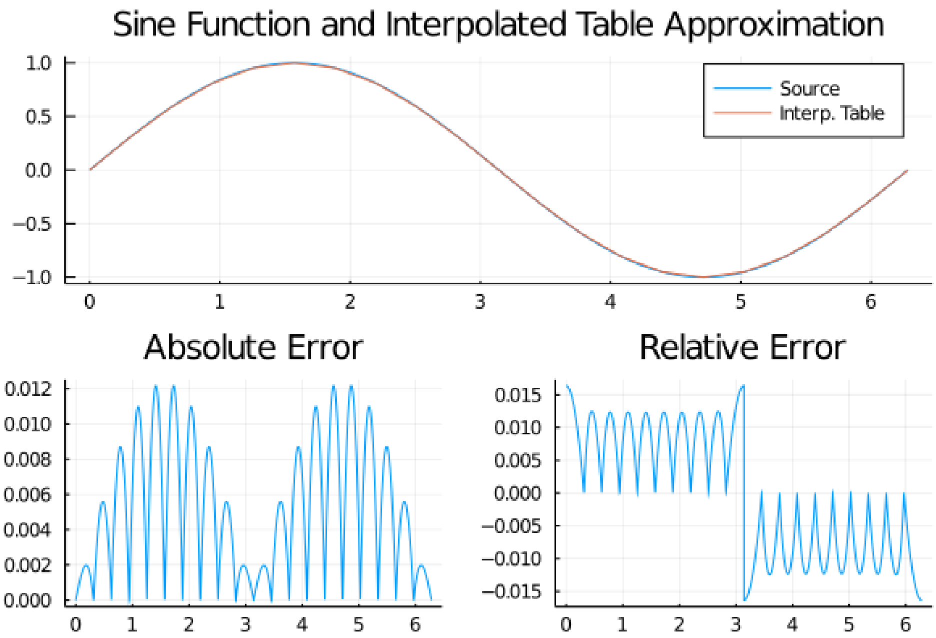
An example of the curve reconstruction from the gradients and its error compared to the correct `sin` function is shown in Figure A.3. This implementation results in further reductions to the overall error. The error is still greatest at the maximum change of gradient because the correction with the gradients is not complex enough to be exact. This approach gives 10x less absolute error than direct interpretation and 15x better than a direct table lookup. In relative error, it is 5x better than interpolation alone and orders of magnitude better than direct lookup.

This is just the beginning of what can be done to improve the accuracy of table-based function implementations. Further improvements can be found to the worst case and average case error. The number of samples can be computed to find the minimum error, the sample steps can be changed to be non-linearly spaced to improve precision at positions of high change of gradient, the sample positions can be shifted to find the global minimum etc.

There is a lot that can be done here! This simple implementation alone gives the worst-case 3% error with only 20 samples. If you wanted to take this further, note that we are using 8 bytes per table entry, which can be reduced if you use 16-bit floats for the gradient to 6 bytes. A typical modern cache line is 64-bytes. The instruction and data cache are separate, so the gradients and constants can be separated to be partially stored as constants in the instructions. This gives a lot of flexibility if you want to create a very fast table based `sin` function with minimal memory access.

A.1.2 Polynomial Approximation

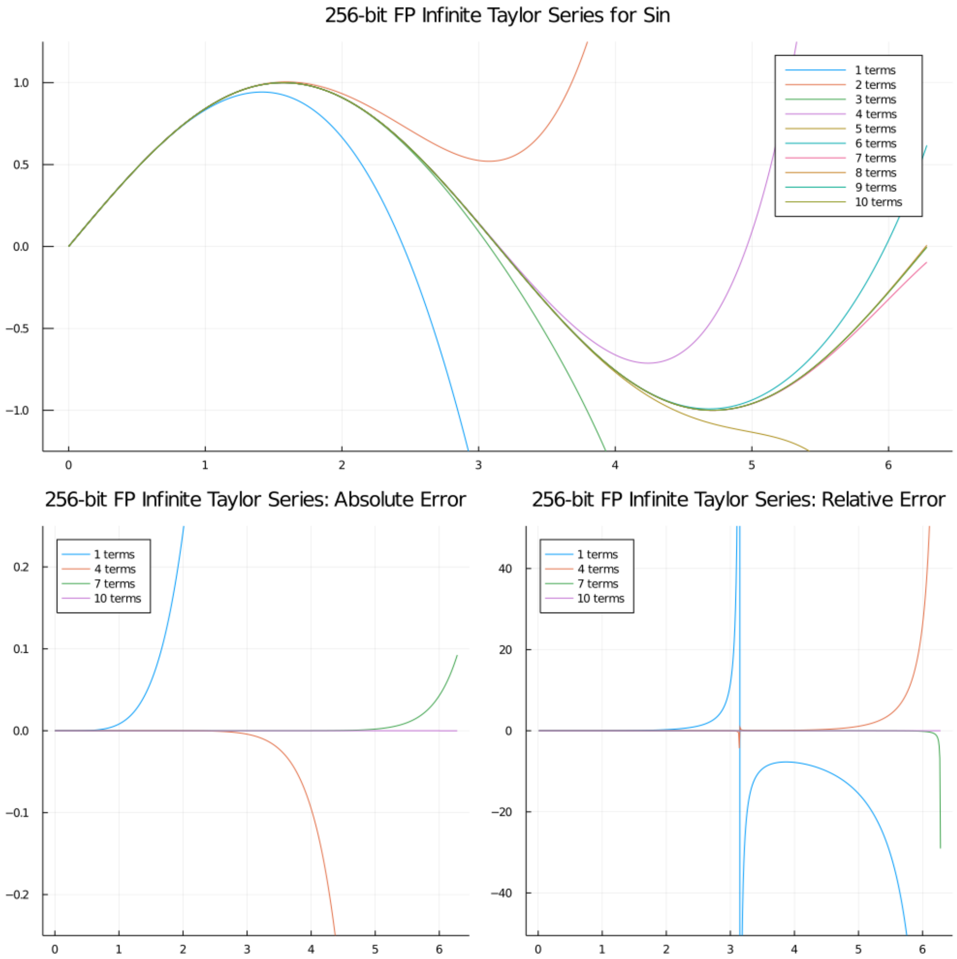
This section gives an example and explanation of different polynomial approaches to approximating a function for a given domain, and then concludes with a brief explanation of the error and accuracy considerations of their implementation.



LOOKUP TABLE (n=20)	
`sin` *Input*	*Output*
0.0	0.0
0.3141...	0.3090...
0.6283...	0.5877...
0.9424...	0.8090...
1.2566...	0.9510...
⋮	⋮

$$
\begin{aligned}
k &= \text{floor}(x/\text{step}) \\
\text{idx}_0 &= k \\
\text{idx}_1 &= \text{idx}_0 + 1 \\
j &= (x - k * \text{step})/\text{step} \\
y &= ((1-j) * \text{table}[\text{idx}_0]) + ((j) * \text{table}[\text{idx}_1])
\end{aligned}
$$

Figure A.2: An example of a small table of n=20 input-output pairs for `sin` used for interpolated lookup.



LOOKUP TABLE (n=20)		
`Sin` *Input*	*Output*	*Gradient*
0.0	0.0	1.0
0.3141...	0.3090...	0.9511...
0.6283...	0.5877...	0.8090...
0.9424...	0.8090...	0.5878...
1.2566...	0.9510...	0.3090...
⋮	⋮	⋮

$$\begin{aligned}
k &= \text{floor}(x/\text{step}) \\
\text{idx}_0 &= k \\
\text{idx}_1 &= \text{idx}_0 + 1 \\
j &= (x - k * \text{step})/\text{step} \\
pred_0 &= (\text{gradients}[\text{idx}_0] * j) + table[idx_0] \\
pred_1 &= (\text{gradients}[\text{idx}_1] * j) + \text{table}[\text{idx}_1] \\
y &= ((1-j) * \text{pred}_0) + ((j) * \text{pred}_1)
\end{aligned}$$

Figure A.3: An example of a very small interpolated table of input-output pairs with the gradient at that position stored for the function `sin`.

A.1.2.1 Infinite Series

Many common mathematical functions can be represented by an infinite series:

$$
\begin{aligned}
sin(x) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \\
cos(x) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \\
tan(x) &= x + \frac{x^3}{3} + \frac{2x^5}{15} - \frac{17x^7}{315} + \dots \\
asin(x) &= x + \left(\frac{1}{2}\right)\left(\frac{x^3}{3}\right) + \left(\frac{1\cdot 3}{2\cdot 4}\right)\left(\frac{x^5}{5}\right) + \left(\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\right)\left(\frac{x^7}{7}\right) + \dots \\
& etc.
\end{aligned}
$$

However limitations of storage and precision give rise to the following issues:

1. **Cost-per-term**
   Each step in the series requires a higher power of the input x, which is in turn more expensive to compute than the last.

2. **Finite terms**
   We can only execute a finite number of terms of the Taylor expansion in finite time.

3. **Term Representation**
   Each term in the series is raised to a larger power. Numbers below one will quickly reduce beyond the precision of the format being used, and numbers greater than one will quickly raise to be either unable to be represented or the nearest representable value with be significantly different than the real number—resulting in error.

4. **Constant Representation**
   Looking at the infinite series representation of `sin(x)` the input raised to a power is divided by an increasingly large factorial of an odd number. In any common hardware supported precision, the result of this factorial is very quickly going to be too large to represent. For example, 32-bit unsigned integers can only represent up to 12 factorial. Which limits this approach to six terms if a 32-bit integer is used.

5. **Destructive Arithmetic**
   As the number of terms increase, so does the number of arithmetic operations which will result in some floating-point error for each additional operation.

For these reasons, direct implementations, even higher precision ones with many terms are not suitable for most uses as they are expensive and not accurate enough.

A good example of this can be seen in the 32-bit implementation (Figure A.4a) where the limitation of 32-bit floating-point to represent 12! means that only up to six of the terms with factorials can be evaluated before the next step is not able to be represented. This limits the function from being able to represent the whole $[0, 2\pi]$ range accurately. It could be used as a section of a piece-wise function used to reconstruct `sin` but not directly like the 64-bit (Figure A.4b) or higher implementations can be.

A full working of an approach for correctly rounded table based approach can be found in the work of Shmuel Gal where they created the *Accurate Tables Method* [59, 60, 150] and related work [156].

A.1.2.2 Continued Fractions

Similar to infinite series are continued fractions. They are a way to represent a function as a series of continuing divisions. Any convergent infinite series can be represented as an infinite continued fraction [49].

Here is an example of `sin` as an infinite continued fraction:

$$\sin(x) = \frac{x}{1+\frac{x^2}{2\cdot3-x^2+\frac{2\cdot3x^2}{4\cdot5-x^2+\frac{4\cdot5x^2}{6\cdot7-x^2+\ldots}}}}$$

Continued fractions give similar arithmetic problems as infinite series, but in the case of some functions, the constant values can be more suitable for the precision or data type being used. For example, in the infinite series `sin` the constant denominator for each term limits the number of terms that could be used with a low-precision data type. In the continued fraction form the constants still grow larger with each term but grow at a much slower rate and therefore allowing for more terms. Although, a division is likely to result in higher floating-point arithmetic error than addition, so there is a trade-off being made.

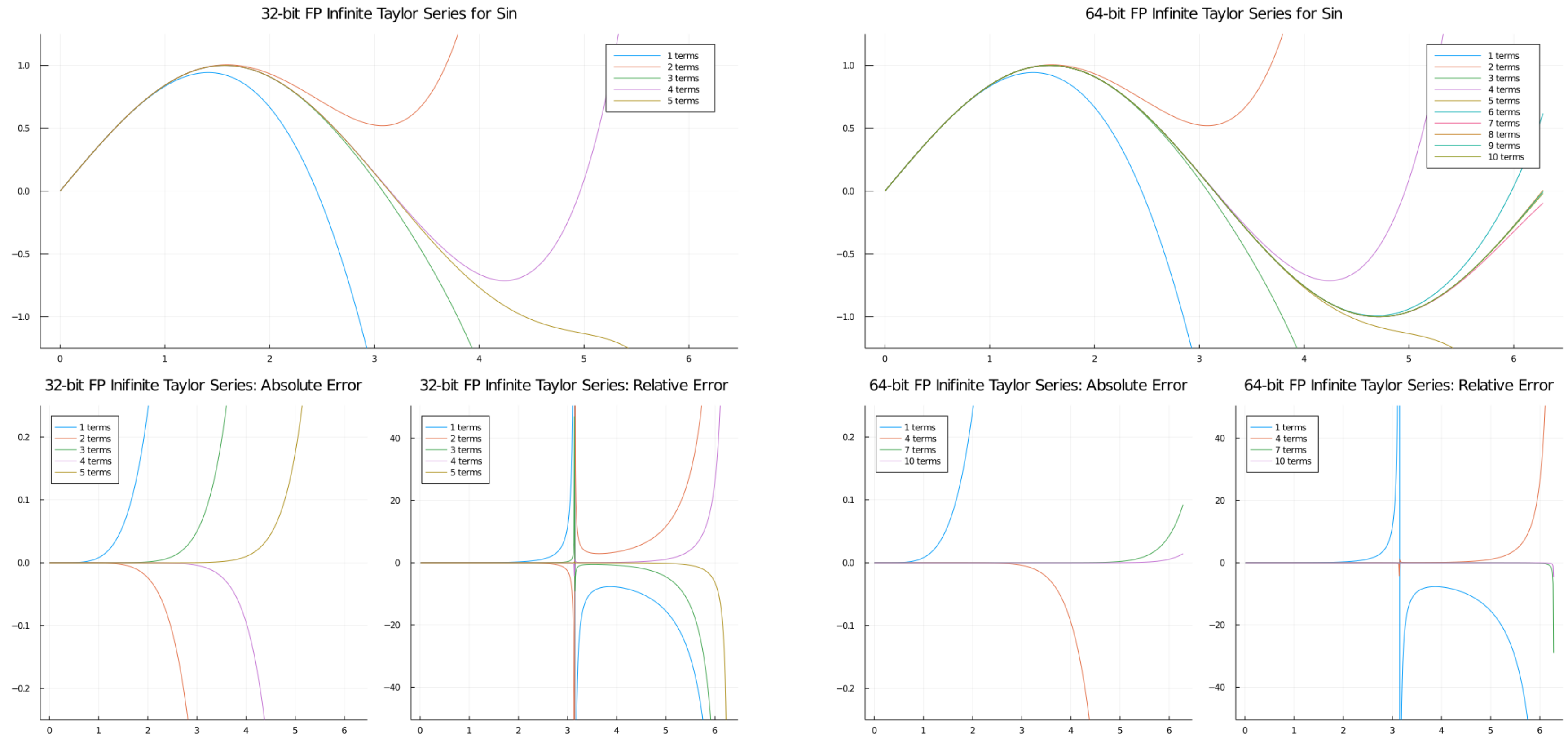
Figure A.5 shows a continued fraction implementation of sin. The resulting curves and error graphs are remarkably similar to those of the infinite series which shows the related nature of the two approaches. Although, for this example, the continued fraction approach is more stable with a higher number of terms and only for large values of `x`.

A.1.2.3 Orthogonal & Minimax Polynomials

Orthogonal polynomial series are a family of polynomials which began with study of continued fractions by P. L. Chebychev [154].

The orthogonal polynomials have an interesting property that is useful for measuring error in approximations:

$$\int w(x)P_i(x)P_j(x)\,dx = \begin{cases} c & \text{if } P_i = P_j \\ 0 & \text{otherwise} \end{cases} \text{ where } w \text{ is a weighting function}$$



(a) The function `sin` implemented in 32-bit with a limited number of terms of the infinite series.

(b) The function `sin` implemented in 64-bit with a limited number of terms of the infinite series.

Figure A.4: An example of an infinite series based `sin`.

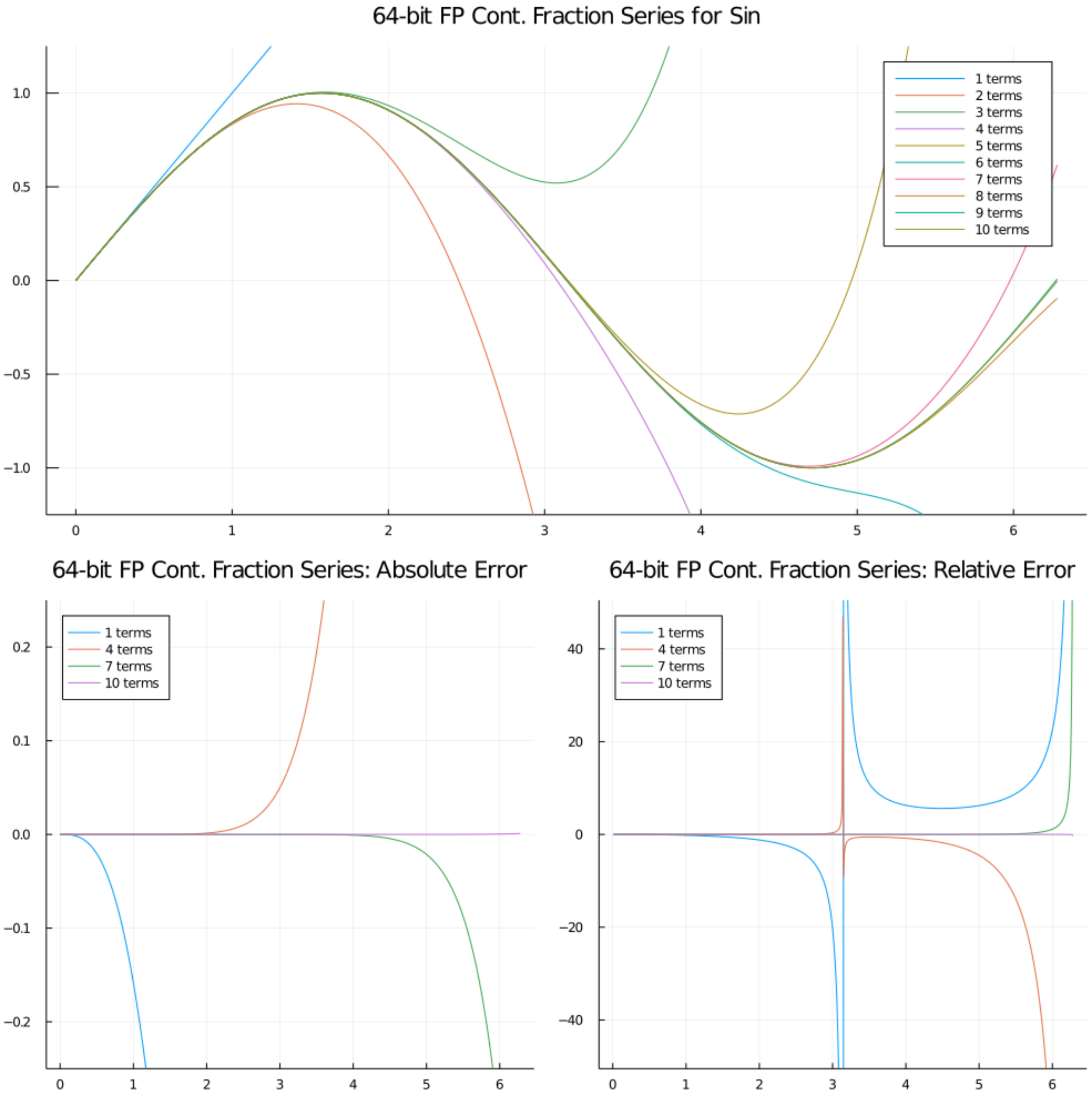
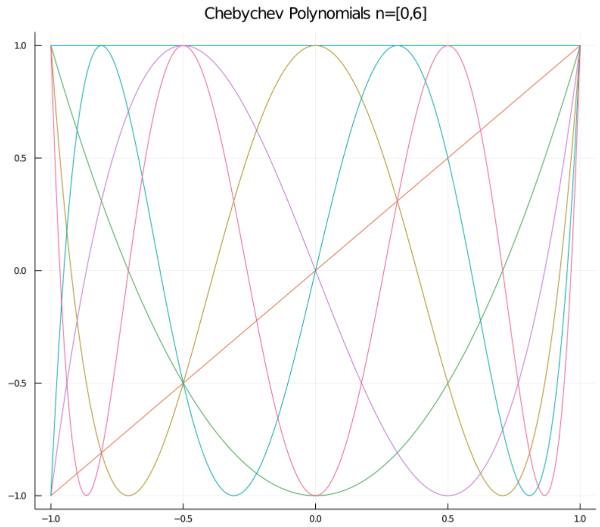


Figure A.5: The function `sin` implemented in 64-bit with a limited number of terms of the continued fraction.



$$
\begin{aligned}
T_0(x) &= 1\\
T_1(x) &= x\\
T_2(x) &= 2x^2 - 1\\
T_3(x) &= 4x^3 - 3x\\
T_4(x) &= 8x^4 - 8x^2 - 1\\
T_5(x) &= 16x^5 - 20x^3 + 5x\\
\\
T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x)
\end{aligned}
$$

Figure A.6: The Chebychev orthonormal polynomials.

The integral of the product of two orthonormal polynomials returns zero if the functions are not the same. This means that if we replace P_j with f, an approximation of P_i, we can compute a value which gives the similarity of P_i to f.

Lets demonstrate this using Chebychev polynomials. To use a Chebychev polynomial, we must first calculate a coefficient k_n for each Chebychev term, T_n. This is done by evaluating the above integral for each n using the Chebychev of the first kind weighting function [164]:

$$w(x) = \frac{1}{\sqrt{1-x^2}}$$

This gives us the coefficient value to multiply by each term. Figure A.6 shows the Chebychev polynomials and how to calculate them for an arbitrary number of terms.

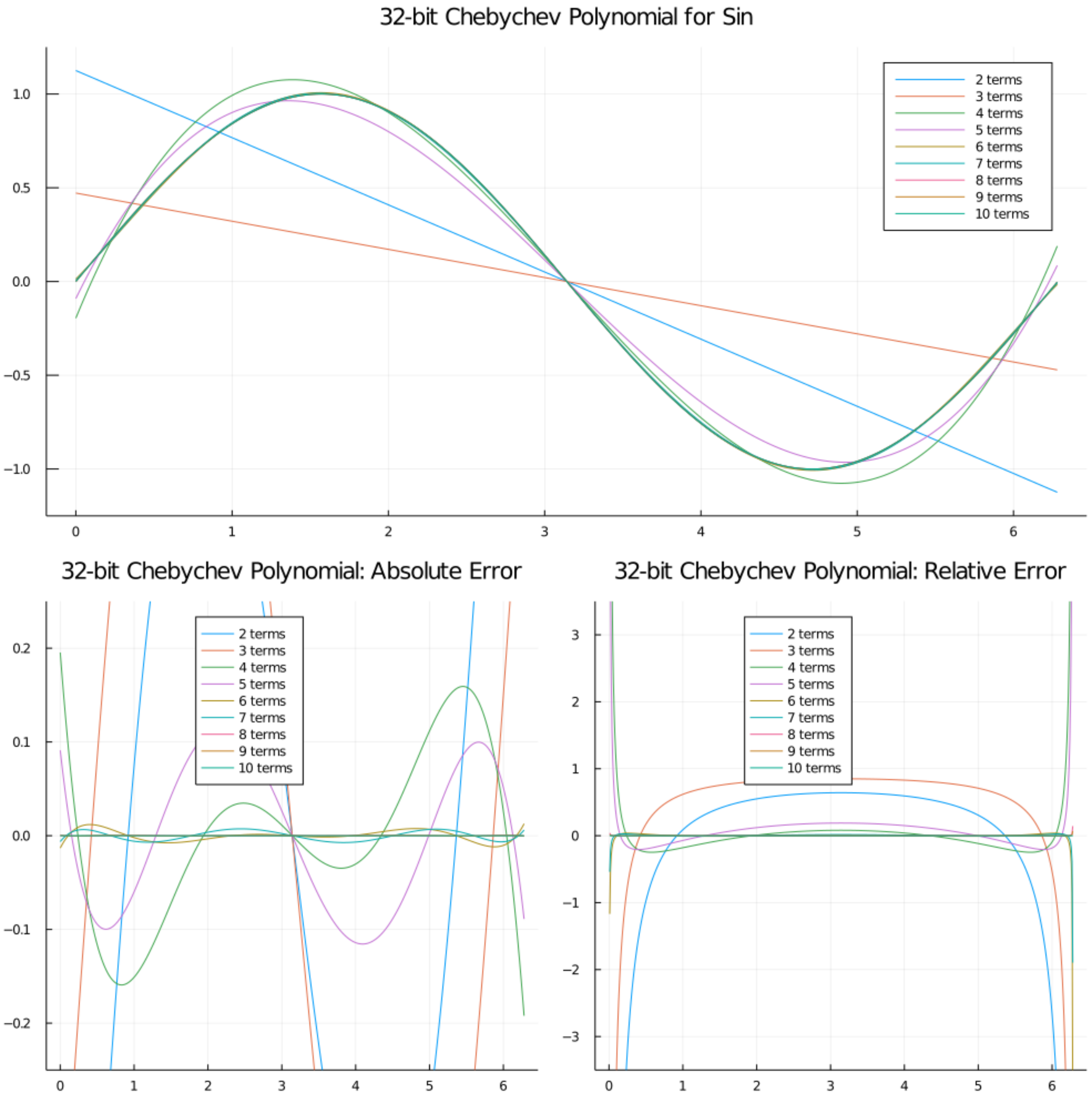
Once the coefficients have been calculated for each term we can construct the approximation. Figure A.7 shows the calculated coefficients and the resulting approximation of the `sin` function using the Chebychev polynomials with 32-bit floating-point numbers.

Like the earlier polynomials, 32-bit suffers from only supporting a limited number of terms before the smaller terms and constants become meaningless. However, you can see that approximations with Chebychev provide very stable minimised error generally, although it has peaks of error at the edge of the chosen domain in this example.

The work of Chebychev in the area of orthogonal polynomials was furthered by Remez [129]. The last algorithm presented in this section is the Remez Algorithm, which is sometimes known as the Minimax algorithm.[1]

The Remez algorithm works by taking an existing polynomial and identifying inputs in the domain where the largest error changes sign. The polynomial weights are then changed to reduce

[1] Despite there being more than one in this category.



Chebychev Coefficients for 6 terms

$$k_0(x) = 2.6597767224800783e-8$$
$$k_1(x) = -0.5692363722260916$$
$$k_2(x) = 8.487245297526617e-8$$
$$k_3(x) = 0.6671667074044133$$
$$k_4(x) = -2.6354203472300778e-8$$
$$k_5(x) = -0.11112301543767747$$

used to approximate sin as:

$$\sin(x) \approx k_n T_n + k_{n-1} T_{n-1} + k_{n-2} T_{n-2} + \ldots + k_1 T_1 + k_0 T_0$$

Figure A.7: An implementation of `sin` for 32-bit floating-point using Chebychev polynomials.

the error at the fixed points. The process is then repeated until the error at the found extremes is reduced below an acceptable threshold.

This can be thought of as iteratively reducing the max error where it is highest and then finding and reducing the new positions of the maximum error until the total error is as low as possible and the remaining error is distributed amongst the terms so that the maximum error is at the lowest possible for a given number of terms.

An example of the results of the Remez algorithm for an implementation of `sin` can be seen in Figure A.8. As it is also based on the orthonormal approach, we can see the error distributions follow a typical orthonormal minimisation pattern of passing through zero between points of maximum error. More details on the implementation of this algorithm can be found in the mathematical textbook by Muller [118], or there are easier to follow code examples in the source code for Remez.jl [24].

A.1.2.4 Performance & Accuracy

On the topic of accuracy, an issue that needs to be addressed is the error occurring from rounding in floating-point. This is the limiting factor for many forms of polynomials due to the large power functions used and can lead to cancellation and imprecision errors between the terms.

There are two levels to approach solving this problem: low-level and algorithmic.

Starting with the low-level we can reduce some error by trying to maintain calculations at a higher-precision before rounding. This is can be done using a FMA(Fused Multiply-Add) operation (covered in more detail in Chapter 3.2.1). The FMA takes arithmetic in the form $-x+y*z$ and performs rounding to the hardware precision after both the multiply and addition operations are complete rather than each add and multiply.

Operations can also be increased in precision and then rounded at the end, for example, $y = \mathrm{f32}(\mathrm{f64}(x)+\mathrm{f64}(x^2)+\mathrm{f64}(x^3))$. However this comes with a memory and, often, a performance penalty. Particularly if the precision is not hardware supported.

For the algorithm level, options include Kahan Summation and Horner's method. Kahan Summation is an approach to summing a mix of large and small floating-point numbers that reduces numerical error. The algorithm works by maintaining a running *compensation* sum of error which is calculated per operation. This compensation is then applied before the next operation.

Horner's method is a way of writing polynomials in code that transforms the regular form we see polynomials written: $a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n$ and changes it to be $(a_0+x(a_1+x(a_2+x(a_3+x(a_4\dots)))))$ [25]. This form of the polynomial is very efficient for computers, especially those which have FMA instructions. It allows degree n polynomials to be evaluated with n multiplications and additions. A significant improvement over treating each term individually.

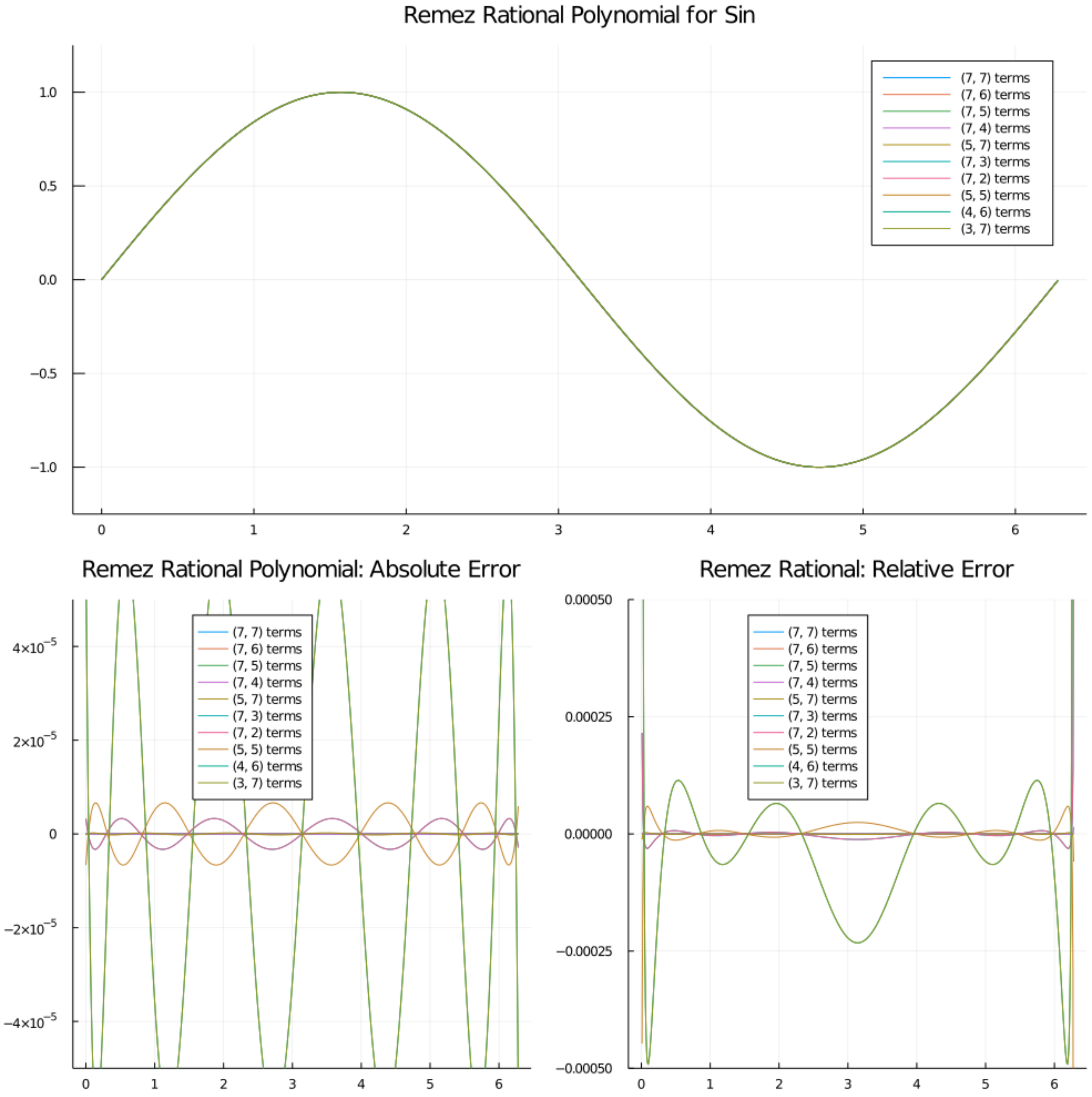


Figure A.8: The function `sin` implemented as a rational polynomial of varying numbers of terms. The error graphs show the expected over/under pattern that is a result of the minimisation of orthonormal polynomials. Each version of the implementation uses a different number of terms for the numerator and denominator. This is denotes as (x,y) in the legend, where x is the degree of the numerator and y is the degree of the denominator.

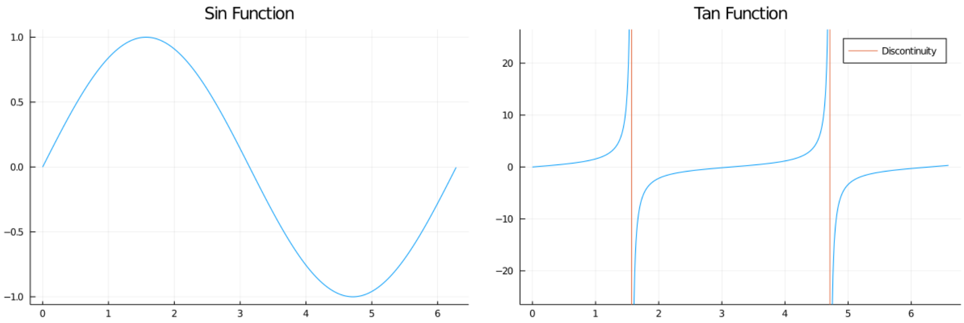


Figure A.9: Showing continuous sin and discontinuous tan.

A.2 Function Properties

A.2.1 Transcendental Functions

If a function is transcendental it means that it cannot be expressed using a finite number of algebraic operations. Which is a little bit of a problem as when using a computer we (mostly) only have access to algebraic operations! This means that it is impossible to ever reach a "correct" evaluation of a transcendental function at infinite precision - any implementation is by definition approximating the true answer. However, at less than infinite precision we can find the closest value to the true answer and return that as the "correctly rounded" result.

A.2.2 Continuous Functions

Continuous functions are those which do not have any discontinuities[2]. This means that any relative change of value will not result in an output for which there is no intermediate value between it and the last value. This is to say that a continuous function is one unbroken curve over the entire domain of the function. With elementary functions, some are continuous (sin/cos/exp) while others are not (tan).

A function being continuous gives it an interesting property which is useful to the evaluation of the function: If a function is continuous and has an output domain between y_1 and y_2 then there is an input x for every representable value in the output y range. If an implementation of a continuous function in a lower precision is not able to output all representable values for that precision then the implementation is no longer "continuous" in its own numerical space and therefore, likely, not a correctly rounded solution.

When considering implementing a continuous function we know that it can be approximated by a polynomial because they are both continuous, but when they are discontinuous special cases and behaviour need to be considered.

[2] I am aware this is like asking "What is a natural disaster?" and answering "A disaster which is natural!" but bear with me on this one...

A.2.3 Periodic Functions

A periodic function is one which repeats. In these cases the functions are often implemented for a small subset of the total range and range reduction is used to set the function input in a relative place with the reduced range.
We can start by considering some different basic categories of error measurement

A.3 Accuracy and Error

A.3.1 Absolute or Relative

The most basic categorisations of error measurements are whether they are absolute or relative.

In this context, absolute error can be computed as the absolute value difference between the result from the approximation and the correct answer, i.e. :

$$\begin{aligned} a_{base} &= f_{base}(x) \\ a_{approx} &= f_{approx}(x) \\ error_{absolute} &= abs(a_{approx} - a_{base}) \end{aligned}$$

Where as relative error is the error relative to some property of the value being evaluated. For example, *ten* with a relative error of *one* gives a 10% error, where as *one hundred* with an error of *one* gives an error of 1%. They both have the same absolute error but it is smaller percentage relative to the value being measured.

$$\begin{aligned} a_{base} &= f_{base}(x) \\ a_{approx} &= f_{approx}(x) \\ error_{relative} &= \frac{abs(a_{approx} - a_{base})}{a_{base}} \end{aligned}$$

We use absolute error in cases where the absolute error difference is critically important and we use relative error when the error difference is important in the context of the result being evaluated.

A good example of when to use absolute or relative error metrics is when using Unit-in-last-place (ULP) error as a measurement of the accuracy of a floating-point algorithm. ULP error is measured as the distance of a floating-point error in increments of the smallest possible change.

This is a relative error metric as the absolute error of the algorithm is divided by the smallest step in floating-point units of the correct output:

$$\begin{aligned} a_{base} &= f_{base}(x) \\ a_{approx} &= f_{approx}(x) \\ error_{ulp} &= \frac{abs(a_{approx} - a_{base})}{eps(a_{base})} \end{aligned}$$

In this example `eps` is a function returning the distance to the next floating-point unit.

Why would we want to use ULP error instead of absolute error for floating-point? Take the function `f(x)` which for an input of 1,000,000 returns 1,000,000.0625. But for an input of 10,000,000 return 10,000,001. If we are judging error in absolute terms we may be tempted to say `f(x)` is more accurate at 1,000,000 than 10,000,000. But this may be an error because when we look at the minimum step size in floating point for the two inputs we see 0.0625 and 1.0 accordingly. So while the output absolute error may be larger, the bits representing the function are each equally incorrect by only one bit. In this case the ULP error at both inputs is the same. The scale of the absolute error is linked to the scale of the output, therefore when it is decoupled it gives a representation of the error that is more useful for analysis.

A.3.2 Minimum, Maximum or Average

After deciding on whether the absolute or relative error is most appropriate for an application, the next decision is how to apply that to more than one data point. Assuming the algorithm being tested has a domain of more than one input then it is necessary to use some means to reason about the results together.

A.3.2.1 Min/Max Error

The two easiest to consider are minimum error and maximum error. If you return these, you are able to tell the user the total scope of the error. For many applications this will be fine, especially when stating maximum error as it is possible to assume “correct” functions to have a minimum error of zero for many cases. It is possible to tell a user that no matter what this algorithm will not exceed a specific value - which is very useful for programmers who care about mapping the worst case through a program.

Because of the usefulness of this metric some mathematical libraries will provide a table stating the maximum ULP error for each of the functions they provide [44, 4].

A.3.2.2 Average Error

If minimum and maximum error-bounds are not enough information to guide how the program can be changed then the distribution of the error must begin to be considered. It might be that the worst case maximum error is very large, but if that worst-case is only one result in a million and the others are all perfectly correct then this might be a case where that singular large error is acceptable. Without investigating the distribution, this would not be possible to assess.

We can start with the simple mean of the error ($sum(e)/length(e)$). If this value is drastically different than our worst-case error then we might begin to suspect that the worst-case error is not representative of the general results.

Next we might consider the 'mode' average. This return the most commonly occurring error. If you are dealing with real numbers then this might not be the best approach, but if you are dealing with ULP error which is often rounded to the nearest whole number to represent the number of steps after rounding, then this might be useful. If the worst-case error is 5 ULP error but the mode is 0 or 1 then plotting frequency of each result might give information about the most common space for error.

Finally, we come to the median result. This is the result which is the middle value of the sorted array of all errors. Finding the median error is a great tool for discounting outliers as it wont be affected by outlying low frequency small or large errors, unlike the mean.

Naturally each of these approaches can be extended to explore the data more thoroughly such as extending the median by also finding the interquartile ranges and other information to provide more insight into the distribution and spread of the error. But at the moment we want to focus on why we might care about the average error when functions are supposed to be accurate.

Average error, in all it's different forms, is useful in the analysis of approximate functions because many algorithms in the real-world are noisy. They produce output with inaccuracies for many reasons ranging from the number systems being used to inaccuracies in the hardware providing some data. In those cases, a sudden large error would likely be ignored and the program could continue on as normal, but if the average error over many iterations was becoming very large then it might hint at a larger problem and the system may not be suitable.

A.3.3 Ranges and Domains of Error

Now that we understand how to assign a meaningful value of error to a result and how to consider that result with regards to other inputs to the same system we next want to consider the domain of those inputs to the function.

It is not uncommon for a function to work for only a limited range of inputs. Take the `sqrt` function for example. If it does not support imaginary numbers then it will only be valid for positive inputs.

When it comes to algorithms which rely on floating-point numbers it is also not uncommon for them to suffer from larger absolute error as the input values diverge from zero, which may be important in a lot of cases. A result of this is that we may only be interested in the error around a certain subsection of the total domain of the function.

Take the fast inverse square root implementation that is presented in Chapter 3. There, input values below 1/3 had significantly higher absolute error than the rest of the input range. As a result we may have an implementation which uses a more accurate approximation when the input is less than 1/3. Therefore, we would only be interested in measuring the error in the specific input sub-domain of $[\frac{1}{3}, \infty]$.

This is particularly relevant in Chapter 6 where we demonstrate how to optimise the implementation of a sin function using approximation. As the implementation of sin is commonly constructed piece-wise from two polynomials, one representing the 'caps' of the curve and another representing the straighter joining edge. Each of these polynomials are only used in very specific sub-domain ranges and as such when we measure the error which our approximations produce we limit the analysis to these ranges.

A.3.4 Deterministic Error and Uncertainty

In computer science error can be deterministic, dependent or stochastic. These needs clarifying so that error in approximate computation is not misunderstood to always be uncertainty.

A program which represents a continuous function using a discontinuous numerical system will have a deterministic sum of error for its domain that is known and constant. This error is not an uncertainty where the error may or may not be present as it would be in engineering when stating the error of a measuring tool, i.e. a basic ruler having uncertainty in its measurements of +/- 1.0mm. It is a constant known value that is a result of the construction of the program representing the continuous function.

However, some error can be bounded by a minimum and maximum error per run of the program and behave like an uncertainty (in which case we may use standard error analysis techniques). An example of this would be a function which polls a sensor on the device. The sensor on the device will have its own uncertainty and tolerance bounds which when polled will produce a value. In this way, the program would have a non-deterministic error measurement that would be based on the uncertainty of the sensor as well as factors arising from the implementation of the program such as numerical accuracy, arithmetic accuracy, compression or other common error inducing features.

Finally, we must mention dependent error. This is error which is dependent on specific hardware, software or environment details which change how a constant program is ran. In these scenarios it can be hard to determine the error (of any category) without full knowledge of the complete setup. This presents itself often through different implementations of standard libraries, different hardware implementation of functions or programs which may change behaviour based on external inputs.

An example of software dependent error would be running a program which links to Microsoft's C standard library and then running it again but linking to a *nix implementation of the same library. Small differences may cause side-effects or minor errors in the program which could result in different outputs.

For hardware dependent error we can consider the implementations of FSIN/FCOS functions in x86/x87 based CPUs. There implementations vary between chips which can lead to different outputs for the same inputs depending on where the program is ran [50].

The environmental dependent error is one which is vaguer and can be trickier to detect. This could refer to thermal interference, solar interference or human interference causing irregularities in the execution of the program and resulting in different outputs. Many of these can be prevented through correct and safe hardware configuration so they will mostly be not considered in this work, but they are worth mentioning as real and important sources of error which need to be considered for some applications.