Proceedings of the 7th Conference on Real Numbers and Computers (RNC'7)

Guillaume Hanrot, Paul Zimmermann

To cite this version:

Guillaume Hanrot, Paul Zimmermann (Dir.). Proceedings of the 7th Conference on Real Numbers and Computers (RNC'7). Guillaume Hanrot, Paul Zimmermann. None., 151 p., 2006, 2-905267-50-X. inria-00107213

HAL Id: inria-00107213

https://inria.hal.science/inria-00107213

Submitted on 17 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Preface

The 7th edition of the “Real Numbers and Computers” conference has been held in Nancy, France from July 10 to July 12, 2006. These proceedings contain all contributed papers presented at RNC’7, together with abstracts of the invited talks by Richard Brent, Stuart Oberman and Vadim Shapiro, and abstracts of the papers accepted in the poster session.

RNC’7 is the seventh of a series of conferences held in Europe. The first edition took place in St-Étienne (France) in 1995, followed by

- RNC’2 in Marseille (France),
- RNC’3 at Université Paris 6, Paris (France),
- RNC’4 in Schloss Dagstuhl (Germany),
- RNC’5 in Lyon (France), and
- RNC’6 in Schloss Dagstuhl (Germany).

The focus of the conference is on computer arithmetic in a rather broad sense, including e.g. computability, geometric algorithms and formal proofs in computer arithmetic. During RNC’7 has been held a friendly competition for multiprecision arithmetic packages, the “Moredigits” competition. This competition has been organized by L. Fousse, V. Lefèvre and N. Müller.

In response to the call for papers, the Program Committee received 22 submissions. Each submission has been anonymously refereed by at least 3 referees. After a revision process by the authors, 12 papers have finally been accepted for presentation at the conference.

The topics addressed by these papers are rather broad. Several topics proved to be especially active by a good number of submissions: hardware design, computability and complexity, efficient implementation of mathematics libraries, representation of numbers, and reliable computations (algorithms and testing). On the other hand, two expanding topics are also represented: arithmetic on GPUs and formalization of real numbers.

We wish to thank all the authors who submitted papers for the conference, the members of the Program Committee and all the external reviewers who helped the Program Committee in its task of selection of the papers.

We acknowledge the financial help of our sponsors: LORIA, INRIA, CNRS, Université Henri-Poincaré Nancy 1, Institut National Polytechnique de Lorraine, Université Nancy 2, Communauté Urbaine du Grand Nancy and Conseil Régional de Lorraine and thank them for their support. We also would like to thank Mairie de Nancy for the reception organized at the town hall.

Finally, we would like to thank all the people who participated in the organization of this conference, especially Anne-Lise Charbonnier (financial aspects, practical organization), the “service communication” from LORIA and Emmanuel Thomé (webmaster, proceedings) who all did a great job.

Let us conclude by incitating everyone to submit papers to the forthcoming Arith’18 conference that will be held in Montpellier in 2007. Hopefully we shall be meeting again there!

Guillaume Hanrot
Paul Zimmermann

RNC’7 co-chairs.
`http://rnc7.loria.fr`

Organisation

RNC'7 has been hosted in LORIA (Laboratoire Lorrain de Recherche en Informatique et Applications), Nancy, France, from July 10th to July 12th, 2006.

Program Committee

Elisardo Antelo	Santiago de Compostela, Spain
Henk Barendregt	Nijmegen, The Netherlands
Vasco Brattka	Cape Town, South Africa
Nicolas Brisebarre	Saint-Étienne, France
Hervé Brönnimann	New York, United States
Martín Escardó	Birmingham, United Kingdom
Guy Even	Tel Aviv, Israël
Christiane Frougny	Paris, France
Guillaume Hanrot	Nancy, France (co-chair)
Peter Kornerup	Odense, Denmark
Paolo Montuschi	Torino, Italy
Norbert Müller	Trier, Germany
Michael Parks	Santa Clara, United States
Siegfried Rump	Hamburg, Germany
Paul Zimmermann	Nancy, France (co-chair)

Steering Committee

Jean-Claude Bajard	Montpellier, France
Vasco Brattka	Cape Town, South Africa
Jean-Marie Chesneaux	Paris, France
Marc Daumas	Montpellier, France
Christiane Frougny	Paris, France
Peter Kornerup	Odense, Denmark (chair)
Dominique Michelucci	Dijon, France
Jean-Michel Muller	Lyon, France
Norbert Müller	Trier, Germany

Organisation Committee

Coordination	Guillaume Hanrot
	Paul Zimmermann
Website, proceedings	Emmanuel Thomé
Friendly competition	Laurent Fousse
	Vincent Lefèvre
	Norbert Müller
Registration, planning	Anne-Lise Charbonnier
	Céline Simon
Poster, publicity	Julian Rivierre
	Bénédicte Maure

Referees

Elisardo Antelo
Henk Barendregt
Andrej Bauer
Valérie Berthé
Vasco Brattka
Nicolas Brisebarre
Herve Brönnimann
Chichyang Chen
Éric Colin de Verdière
Pierre Courtieu
Florent de Dinechin
Martín Escardó
Guy Even
Laurent Fousse
Christiane Frougny
Torbjörn Granlund
Guillaume Hanrot
Ahmad Hiasat
Peter Kornerup
Fabrizio Lamberti
Vincent Lefèvre
Paolo Montuschi
Jean-Michel Muller
Norbert Müller
Russell O'Connor
Michael Parks
Siegfried Rump
Damien Stehlé
Arnaud Tisserand
Xavier Urbain
Liang-Kai Wang
Freek Wiedijk
Paul Zimmermann

Sponsoring Institutions

CNRS (Centre National de la Recherche Scientifique)
INRIA (Institut National de Recherche en Informatique et Automatique)
Université Henri Poincaré – Nancy 1
Université Nancy 2
Institut National Polytechnique de Lorraine
Mairie de Nancy
Communauté Urbaine du Grand Nancy
Conseil Régional de Lorraine

Table of Contents

Part I

Abstracts of Invited Talks

GPUs: Applications of Computer Arithmetic in 3D Graphics (extended abstract)

Stuart F. Oberman

NVIDIA
Santa Clara, California
USA

The rendering of 3D graphics places extreme demands on computational hardware. Dedicated Graphics Processing Units, or GPUs, are designed to enable high performance 3D graphics and multimedia. Modern GPUs incorporate highly-parallel architectures, making them more effective than traditional CPUs for a wide range of complex parallel algorithms, but especially for the challenges relating to image synthesis.

CPUs comprise a relatively small proportion of computational area relative to the area used for caches and instruction issue control. In 2006, a high-end dual-core x86 CPU contains about 150 million transistors, and it has a peak floating-point throughput of approximately 50 single-precision GFLOPS. In contast, a 2006 high-end GPU contains about 300 million transistors, and it has a peak floating-point throughput of nearly 300 single-precision GFLOPS of a traditional, programmable nature. In addition to this general programmable floating-point computational power, GPUs contain a wide variety of special purpose, fixed-function arithmetic, whose purpose it is to accelerate specific components of the 3D graphics pipeline. The total of all of the special purpose arithmetic hardware may exceed an additional several hundred GFLOPS. Some of this special purpose arithmetic operates on floating-point operands, although not necessarily in IEEE 754 compatible formats. Other arithmetic units operate on fixed-point operands. In total, comparing CPUs to GPUs, a current high-end GPU has about twice the transistor count of a high-end CPU, yet it has an order of magnitude more total arithmetic processing power.

In this talk I will provide some history of GPUs. This will include a discussion of the evolution of GPUs from their 2D-only ancestors, to the 3D-only accessory boards, to the modern Graphics Processor, motivated using representative images. A brief overview of the canonical 3D graphics pipeline will then be presented. A typical scene to be displayed is first described by triangles of materials simulated by sampled images and textures. Mathematical processing is performed on each of the triangles' vertices, in order to rotate, translate and scale each object in the scene, as well as to properly adjust the entire scene's position in the effective camera's field of view. This process of vertex processing is accomplished using *vertex shaders*. Processed triangles are then rasterized into pixels. The rasterization process involves identifying every pixel that belongs to a triangle. Several arithmetic units are required to perform the various tasks implicit in triangle rasterization. Finally, the pixels themselves are processed using *pixel shaders*. This pixel computation may include texture sampling, color calculation, and various blending functions.

I will then discuss the evolution of *shaders* from fixed-function hardware to fully programmable computational engines. The microarchitectures of modern GPUs will be explored to understand how each step of the graphics pipeline may map into either fixed-function arithmetic units or programmable shaders. I will then discuss in more detail some of the arithmetically intensive operations, including the core shading engines. The computational core of a modern shader will be examined, including floating-point multiplication, addition, and multiply-add units, as well as the requirements and algorithms used for the various higher-order functions, such as reciprocal, reciprocal square-root, binary logarithm, binary exponential, and the sin and cos functions. Other shader arithmetic aspects to be examined include the details of per-pixel attribute interpolation and its mapping to arithmetic hardware. Finally, the details of texture sampling and filtering will be examined. Given a texture residing in memory, the usage of MIP maps will be discussed. The arithmetic algorithms used for finding the appropriate level-of-detail in a MIP map stack for a given pixel will be investigated, followed by a look at the arithmetic required for visually acceptable texture sample filtering.

Tolerancing and Metrology of Geometric Models (extended abstract)

Vadim Shapiro[⋆]

Mechanical Engineering and Computer Sciences
University of Wisconsin - Madison
vshapiro@engr.wisc.edu

Computer representations of geometric models of physical artifacts have become the principal means for creating, communicating, exchanging, and analyzing engineering information. Geometric modeling is supported by several elegant mathematical theories that assume the ability to compute and store geometric representations exactly. In contrast, all practical implementations rely on real number computations with finite computational resources. This incompatibility between the theories and their implementations led to emergence of new academic problems, broadly referred to as geometric robustness and tolerant modeling, that remain without satisfactory solutions. It is also a cause of great practical difficulties with substantial negative economic impact that is measured in billions of dollars annually. A whole new industry of geometric validation and repair has been created in an attempt to circumvent these fundamental challenges created by mismatch between the theory and practice. Several international standards are being created for cataloguing and categorizing both difficulties and ad hoc solutions to geometric data quality problems.

Important lessons can be learned by treating geometric models as manufactured objects, similar to traditional mechanical parts and assemblies. It is generally accepted that modern mass production and most of the manufacturing technologies of the past century could not have been possible without the doctrine of *interchangeability.* It dictates that a mechanical part may be replaced by another 'equivalent' component without affecting the overall function of the product. Prior to the adoption of this principle, manufacturing was a custom art practiced mainly by skilled artisans who made all components to nominal size, as precisely as possible, and on a very small scale. Implementation of the principle of interchangeability led to standardized principles of *tolerancing* (focused on specification and control of geometric variability) and *metrology* (inspection of manufactured parts through measurement and analysis of accuracy). Recent efforts focus on mathematical foundations of geometric dimensioning and tolerancing, using concepts of tolerance *zones* and material *containments* conditions.

I will argue that parallel efforts on tolerancing and metrology are needed to reconcile the theory of geometric modeling with the computational reality. The theory should recognize that modeling exactly, or as precisely as possible, is not a viable practical alternative. It is reasonable to expect that a more general theory of geometric modeling should include concepts of interchangeability, tolerances, zones, and containment, but it is also important that this theory contains the classical exact theory as a special case. An important difference between tolerances in manufacturing and in geometric modeling is that manufacturing accuracy concerns exclusively with variability of real *physical* objects, whereas geometric errors can easily render a *computer* geometric model non-physical, and therefore invalid. Establishing validity conditions for a toleranced geometric model is one of the key problems in metrology of geometric models. I will propose the beginnings of such a new theory for tolerancing and metrology of geometric models, but many challenging questions remain open.

[⋆] Supported in part by National Science Foundation grants OCI-0537370, DMI-0500380, DMI-0323514

Fast Algorithms for High-Precision Computation of Elementary Functions (extended abstract)

Richard P. Brent

Australian National University
http://www.rpbrent.com/

In many applications of real-number computation we need to evaluate elementary functions such as $\exp(x)$, $\ln(x)$, $\arctan(x)$ to high precision (see for example [1]). We shall survey some of the well-known (and not so well-known) techniques as well as mentioning some new ideas.

Let d be the number of binary digits required, so the computation should be accurate with relative (or, if appropriate, absolute) error $O(2^{-d})$. By "high-precision" we mean higher than can be obtained directly using IEEE 754 standard floating-point hardware, typically d several hundred up to millions.

We are interested both in "asymptotically fast" algorithms (the case $d \to +\infty$) and in algorithms that are competitive in some range of d. Let $M(d)$ denote the time (measured in word- or bit-operations) required to multiply d-bit numbers with d-bit accuracy (we are generally only interested in the upper half of the $2d$-bit product). Classically $M(d) = O(d^2)$ and the Schönhage-Strassen algorithm [12] shows that $M(d) = O(d \log d \log\log d)$. However, Schönhage-Strassen is only useful for large d, and there is a significant region $d_1 < d < d_2$ where A. Karatsuba's $O(d^{\lg 3})$ algorithm [8] is best ($\lg 3 = \log_2 3 \approx 1.58$). In the region where Karatsuba's algorithm is best for multiplication, the best algorithms for elementary functions need not be those that are asymptotically the fastest.

Sometimes the best algorithm depends on the ground rules: are certain constants such as π allowed to be precomputed, or does the cost of their computation have to be counted every time in the cost of the elementary function evaluation?

Techniques for high-precision elementary function evaluation include the following. Often several are used in combination, e.g. argument reduction is used before power series evaluation.

1. Argument reduction using identities such as

$$\exp x = (\exp(x/2))^2 \ , \ \ \arctan x = 2 \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \ , \ \ \text{etc.}$$

2. Use of power series such as

$$\exp x = \sum_{k\ge 0} \frac{x^k}{k!} \ , \ \ \ln(1+x) = \sum_{k\ge 0} \frac{(-1)^k x^{k+1}}{k+1} \ , \ \ \arctan x = \sum_{k\ge 0} \frac{(-1)^k x^{2k+1}}{2k+1} \ ,$$

perhaps evaluated using the technique of Smith [13], which applies more generally (for example to the evaluation of hypergeometric functions). Smith seems to be the first to apply his technique for real computation, but the idea was suggested by Paterson and Stockmeyer [10] and used in a different context by Brent and Kung [6] (but not in the author's multiple-precision package [5], because of the storage requirements).
By combining argument reduction and power series evaluation we get an $O(M(d)d^{1/2})$ algorithm for $\exp(x)$, and using Smith's technique this can be improved to $O(M(d)d^{1/3}) + O(d^{5/3} \log\log d \log\log\log d)$ (the second term is essentially $O(d^{5/3})$ in practice).

3. Use of the arithmetic-geometric mean (AGM) to compute $\ln x$ in time $O(M(d)\log d)$ (asymptotically the fastest known), see [2–4]. In particular we mention the algorithm of Sasaki and Kanada [11], based on the elegant formula

$$\ln x = \frac{\pi}{\mathrm{AGM}(\theta_2^2(1/x), \theta_3^2(1/x))} \; .$$

   Because the theta functions have rapidly-converging series and this formula is exact, we can use it for smaller x than is possible with the usual "approximate" AGM-based formulae such as $\ln x = \pi/((2+O(1/x^2))\mathrm{AGM}(1,4/x))$.
4. Use of Newton's method to compute inverse functions, for example we can compute $\exp(x)$ from $\ln(x)$ and *vice versa.* The overhead introduced by Newton's method can be reduced to a factor $1+o(1)$ as $d \to +\infty$ by using higher-order methods [3, §6-§9].
5. Use of complex arithmetic to compute a real result, for example

$$\arctan x = \frac{1}{2i}\ln\left(\frac{1+ix}{1-ix}\right) = \Im \ln(1+ix) \; ,$$

   where the complex log can be computed by the AGM; this gives the asymptotically fastest known algorithm for arctan (although the complex arithmetic is a significant overhead).
6. Use of *binary splitting* [2, p. 329] (or similarly E. Karatsuba's *FEE method* [9]) to sum series with rational arguments [7]. For real arguments, we may be able to use a good rational approximation and then apply a small correction. To illustrate this we shall describe some new ideas for arctan evaluation which, although not asymptotically the fastest, are competitive for a wide range of precisions d (this is joint work with Jim White).

References

1. D. H. Bailey, High-precision floating-point arithmetic in scientific computation, *Computing in Science and Engineering*, May–June 2005, 54–61; also report LBNL–57487. Available from `http://crd.lbl.gov/~dhbailey/dhbpapers/` .
2. J. M. Borwein and P. B. Borwein, *Pi and the AGM*, Monographies et Études de la Société Mathématique du Canada, John Wiley & Sons, Toronto, 1987.
3. R. P. Brent, Multiple-precision zero-finding methods and the complexity of elementary function evaluation, in *Analytic Computational Complexity* (edited by J. F. Traub), Academic Press, New York, 1975, 151–176. Available from `http://wwwmaths.anu.edu.au/~brent/pub/pub028.html` .
4. R. P. Brent, Fast multiple-precision evaluation of elementary functions, *J. ACM* **23** (1976), 242–251.
5. R. P. Brent, Algorithm 524: MP, a Fortran multiple-precision arithmetic package, *ACM Trans. Math. Software* **4** (1978), 71–81.
6. R. P. Brent and H. T. Kung, Fast algorithms for manipulating formal power series, *J. ACM* **25** (1978), 581–595.
7. X. Gourdon and P. Sebah, Numbers, constants and computation: binary splitting method, `http://numbers.computation.free.fr/Constants/Algorithms/splitting.html` .
8. A. Karatsuba and Y. Ofman, Multiplication of multidigit numbers on automata (in Russian), *Doklady Akad. Nauk SSSR* **145** (1962), 293–294. English translation in *Sov. Phys. Dokl.* **7** (1963), 595–596.
9. E. A. Karatsuba, Fast evaluation of transcendental functions (in Russian), *Probl. Peredachi Inf.* **27**, 4 (1991), 87–110. English translation in *Problems of Information Transmission* **27** (1991), 339–360. See also `http://www.ccas.ru/personal/karatsuba/faqen.htm` .
10. M. S. Paterson and L. J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials, *SIAM J. Computing* **2** (1973), 60–66.
11. T. Sasaki and Y. Kanada, Practically fast multiple-precision evaluation of $\log(x)$, *J. Inf. Process.* **5** (1982), 247–250. See also [2, §7.2].
12. A. Schönhage and V. Strassen, Schnelle Multiplikation Grosser Zahlen, *Computing* **7** (1971), 281–292.
13. D. M. Smith, Efficient multiple-precision evaluation of elementary functions, *Math. Comp.* **52** (1989), 131–134.

Part II

Contributed Talks

Fast, Guaranteed-Accurate Sums of Many Floating-Point Numbers

Yong-Kang Zhu and Wayne Hayes

School of Information and Computer Science
University of California,
Irvine, CA 92697
{yongkanz,wayne}@ics.uci.edu

Abstract. We present an algorithm for computing the faithfully-rounded sum of an array of n floating-point numbers. It requires no extended accumulator, and works in any base. Similar to the fastest recently published accurate approaches, its running time is about six times that of the naïve one-loop approach. We prove that it always produces a faithfully rounded result (modulo intermediate overflow), independent of both n and the condition number of the sum, as long as a sufficiently large temporary storage array is available. We argue, and observe empirically, that the maximum size of the temporary array is bounded by a small constant times the maximum number of non-overlapping mantissas that are representable by the machine arithmetic (39 in the case of IEEE754 `double`), although we have not yet proven this bound rigorously.

Keywords: floating-point summation, rounding error, distillation

1 Introduction

The summation of n floating-point numbers, $\sum_{i=1}^{n} x_i$, is ubiquitous in numerical computations, and has been the subject of much recent work. Anderson [1] proposes a distillation algorithm as defined in [3], which iteratively and accurately deflates pairs of oppositely signed floating-point numbers until all of them have the same sign except for those with negligibly small absolute value. The final result is obtained by using compensated summation [4]. Comparisons between this method and other ones can be found in [6, 9].

A new distillation algorithm for floating-point summation was presented in [9], which is more robust and faster than Anderson's method. It adds two summands repeatedly, without discarding any significant digit until the partial sums cannot change the whole sum. It does not rely on the choice of radix or any other specific assumption. Furthermore, its error bound ($\leq$1ulp) is independent of n and the condition number, $R = \sum_{i=1}^{n} |x_i| / |\sum_{i=1}^{n} x_i|$. However, it is significantly slower than some other recently proposed methods.

Ogita et al. [6] presented a method called SumK, which has an integer parameter K, representing a limit on the running time of the algorithm. For K large enough, the algorithm is guaranteed to produce a correct result, but K must increase both with the number of summands n, and the condition number, in order to produce a guaranteed exact result. The authors suggest a value of $K = 3$ for practical purposes, but then the algorithm is guaranteed to fail for certain inputs. Rump et al. [8] introduce a new algorithm named AccSum which works only in binary but whose accuracy is independent of the condition number, although it is still dependent upon n.

In this paper, a new fast algorithm for floating-point summation is presented. Our method guarantees that the result is faithfully rounded.

Its accuracy is independent of both n and the condition number. Its typical running time is almost as fast as the method Sum3 [6], which is the $K = 3$ version of SumK algorithm. However, in very rare cases its running time can be dependent on n and the condition number. In the next section, we describe this method in detail and analyze its error bound. Comparisons are given in section 3. Some further discussion is provided in Section 4. Conclusions are drawn in section 5.

2 Algorithm and analysis

2.1 Algorithm

The accuracy of the method of Zhu et al. [9] is independent of n and the condition number. It uses two arrays to save the positive and negative numbers separately. The operations of adding and removing elements to the array are relatively expensive.

Therefore, it runs slower than the methods described in [6, 8]. Our new method improves upon [9] by pre-allocating an array for saving the terms comprising two temporary sums, s_p and s_n, in each loop. All operations are done in the given arrays. So, no elements need to be inserted or removed from the array.

Our method depends on the existence of the accurate addition between two floating-point numbers. Here, we assume $\{s, e\} \leftarrow \text{AddTwo}(a, b)$ is such an algorithm that $s + e = a + b$ and $s = \text{fl}(a + b)$ (and so the mantissas of s and e do not overlap). There are several choices, such as TwoSum [6], Algorithm2 [9], and Algorithm2′ [9]. In our numerical tests, Algorithm2′ [9] is chosen, since this method is fast and accurate when using binary floating-point arithmetic. It contains only three standard floating-point additions. If used for other bases, then it can be replaced by other accurate algorithms. We also assume that the floating-point operations, $\text{fl}(a \pm b)$, are faithfully rounded [7]. That is, let $y = \text{fl}(x)$ be the operation that returns the floating-point value of x, and $|a| \leq |x| < |b|$, where a and b are two consecutive floating-point numbers with the same sign as x. Here, y is called *faithfully rounded* if and only if $y = a$ whenever $x = a$ and either $y = a$ or $y = b$ whenever $x \neq a$. (Note that this is not as stringent as round-to-nearest, which we do not guarantee.) Our algorithm also makes no effort to avoid intermediate overflow in the case that the sum is representable. The algorithm is described below. We use β to represent the base of the floating-point number, and t to represent the length of the mantissa (for example, in IEEE754 `double`, $\beta = 2$ and $t = 53$).

```
ALGORITHM: $s = \text{FastSum}(x, n, q, L, c_q)$.
    Input:  $x$, the array of the given floating-point summands;
            $n$, the number of summands;
            $q$, the accessory array for recording the temporary sums;
            $L$, the maximal size of the array $q$ (in IEEE754 *double*, set $L = 200$);
            $c_q$, the current number of elements in $q$ (initially 0).
    Output: $s$, the sum with a faithful rounding.

1. $s \leftarrow 0$; $loop \leftarrow 1$;
   //$loop$ counts the number of $n$-sized loops
2. for $i \leftarrow 1$ to $n$
   (1) $\{s, x[i]\} \leftarrow \text{AddTwo}(s, x[i])$;
3. loop forever
```

```
   (1) $count \leftarrow n$; $s_p \leftarrow 0$; $s_n \leftarrow 0$; $loop \leftarrow loop + 1$;
       //$count$ records the number of non-zero numbers remaining in the $x$ and $q$ arrays
       //$s_p$ and $s_n$ accumulate the positive and negative summands respectively
   (2) for $i \leftarrow 1$ to $n$
       (a) if $x[i] > 0$, then $\{s_p, x[i]\} \leftarrow$ AddTwo($s_p, x[i]$);
       (b) else $\{s_n, x[i]\} \leftarrow$ AddTwo($s_n, x[i]$);
       (c) if $x[i] = 0$, then $count \leftarrow count - 1$;
   (3) for $i \leftarrow 1$ to $c_q$
       //notice if $c_q < 1$, then do nothing here and go to Step 3(4)
       (a) if $q[i] > 0$, then $\{s_p, q[i]\} \leftarrow$ AddTwo($s_p, q[i]$);
       (b) else $\{s_n, q[i]\} \leftarrow$ AddTwo($s_n, q[i]$);
       (c) if $q[i] \neq 0$, then $count \leftarrow count + 1$;
   (4) $e_m \leftarrow count \cdot \mathrm{ulp}(\max(|s_p|, |s_n|))$;
       //a weak upper bound on sum of remaining terms in $x$ and $q$
   (5) $e_1 \leftarrow s_p$; $e_2 \leftarrow s_n$;
   (6) do
       (a) $\{e_1, e_2\} \leftarrow$ AddTwo($e_1, e_2$);
       (b) $\{s, e_1\} \leftarrow$ AddTwo($s, e_1$);
       while ($\mathrm{fl}(s + \mathrm{fl}(e_1 + e_2)) \neq s$);
   (7) $c_q \leftarrow c_q + 1$; $q[c_q] = e_1$;
   (8) $c_q \leftarrow c_q + 1$; $q[c_q] = e_2$;
   (9) if $c_q >= L - 1$, then return $s$;
       //error condition, $L$ is set too small
  (10) if $\mathrm{fl}(s + e_m) = s$, then
       (a) $\{E_1, E_2\} \leftarrow$ AddTwo($e_1, e_2$);
       (b) if $\mathrm{fl}(\mathrm{fl}(E_1 + \mathrm{fl}(E_2 + e_m)) + s) = s$ and $\mathrm{fl}(\mathrm{fl}(E_1 + \mathrm{fl}(E_2 - e_m)) + s) = s$, then
           return $s$;
       (c) else return $s +$ FastSum($x, n, q, L, c_q$);
4. END
```

Given the array x and the number of summands n, we first initialize an empty array q whose size is L, and then call FastSum(x,n,q,L,0) to compute the result s. In our algorithm FastSum, Step 2 makes an ordinary iterative summation, saving the errors in x. Step 3 is an infinite loop until the ending conditions (see Steps 3(9) and 3(10)) are satisfied. This loop works as follows. In Step 3(2), all the positive numbers are summed into s_p, while all the negative numbers are summed into s_n. Then in Step 3(3), we perform the same operation on the accessory array q. An upper bound on the sum of all the remaining numbers in x and q is computed in Step 3(4), according to the current temporary sums s_p and s_n. In Step 3(6), s_p and s_n are added to s and in Steps 3(7) and 3(8) the errors are stored into the accessory array q.

2.2 Proof of correctness

Theorem. *If FastSum does not return from Step 3(9), then it generates a result that is faithfully rounded.*

Proof. Since AddTwo gives an exact addition between two floating-point numbers, all additions use AddTwo, and all the errors of AddTwo are stored, the algorithm FastSum does not discard any significant digits before it returns a result. All the errors are stored in the arrays x and q. Assume each floating-point number is represented as

$$x = \pm 0.d_1 d_2 \ldots d_t \times \beta^{\exp(x)}, \tag{1}$$

where $\exp(x)$ is the exponent of the floating-point number x. Let $\mathrm{ulp}(x) = \beta^{\exp(x)-t}$. For each $\{s, e\} \leftarrow \mathrm{AddTwo}(a, b)$, we have $|e| < \mathrm{ulp}(s)$ since $s = \mathrm{fl}(a + b)$ with a faithful rounding. Therefore, after Steps 3(2) and 3(3) we have

$$\begin{aligned} |x[i]| &< \mathrm{ulp}\left(\max(|s_p|, |s_n|)\right), \quad i = 1, 2, \cdots, n, \\ |q[j]| &< \mathrm{ulp}\left(\max(|s_p|, |s_n|)\right), \quad j = 1, 2, \cdots, c_q. \end{aligned} \tag{2}$$

Recalling that *count* is the number of non-zero terms in the x and q arrays, we have

$$\begin{aligned} \left| \sum_{i=1}^{n} x[i] + \sum_{j=1}^{c_q} q[j] \right| &\le \sum_{i=1}^{n} |x[i]| + \sum_{j=1}^{c_q} |q[j]| \\ &< \mathit{count} \cdot \mathrm{ulp}\left(\max(|s_p|, |s_n|)\right) \\ &= e_m. \end{aligned} \tag{3}$$

It can be shown (using Lemma A from Appendix A) that Step 3(6) will stop after at most two loops, at which time $|e_1 + e_2| < \mathrm{ulp}(s)$.

The algorithm may return a result at one of three positions: Steps 3(9), 3(10)(b), and 3(10)(c). We prove the correctness of the algorithm by cases. We disregard the first case since we believe we can set L large enough so that the algorithm never returns from Step 3(9).

Case 1: if s is returned from Step 3(10)(b). When the condition in Step 3(10) is satisfied, we have

$$|e_m| < \mathrm{ulp}(s). \tag{4}$$

Let $x[i]^{(0)}$ be the original summands, and let $x[i]$ be the remaining errors in the array x. Then,

$$\sum_{i=1}^{n} x[i]^{(0)} = s + E_1 + E_2 + \sum_{i=1}^{n} x[i] + \sum_{j=1}^{c_q-2} q[j]. \tag{5}$$

Note that Steps 3(7) and 3(8) are already executed at Step 3(10)(b). So, the last two numbers in q are e_1 and e_2, which are not counted when calculating e_m. To prove s is a faithful rounding of $\sum_{i=1}^{n} x[i]^{(0)}$, our task is to prove

$$\mathrm{fl}\left(s + E_1 + E_2 + \sum_{i=1}^{n} x[i] + \sum_{j=1}^{c_q-2} q[j] \right) = s. \tag{6}$$

Therefore, according to (3), if both

$$\begin{aligned} \mathrm{fl}(s + E_1 + E_2 + e_m) &= s, \text{ and} && (7) \\ \mathrm{fl}(s + E_1 + E_2 - e_m) &= s, && (8) \end{aligned}$$

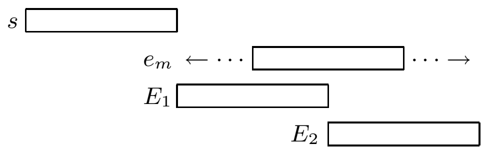


Fig. 1. Schematic illustration of the relationship between s, E_1, E_2, and e_m after Step 3(10)(a). s, E_1, and E_2, have no mutual overlap, but e_m may appear anywhere "below" s.

are satisfied, then we can obtain (6). We will show the ending condition in Step 3(10)(b)

$$\mathrm{fl}(s + \mathrm{fl}(E_1 + \mathrm{fl}(E_2 + e_m))) = s, \tag{9}$$

$$\mathrm{fl}(s + \mathrm{fl}(E_1 + \mathrm{fl}(E_2 - e_m))) = s, \tag{10}$$

imply (7) and (8). From Steps 3(6) and 3(10), we have that $\mathrm{fl}(s+\mathrm{fl}(E_1+E_2)) = s$, $\mathrm{fl}(E_1+E_2) = E_1$, and $\mathrm{fl}(s + e_m) = s$ (see Figure 1). Since $e_m \geq 0$, without loss of generality, we can assume $E_1 > 0$, then (10) clearly implies (8). [To see this, consider (10): (i) If $E_2 \geq 0$, then (10) and (8) are both satisfied due to the cancellation of E_1 and e_m. (ii) If $E_2 < 0$, then E_1 and e_m either overlap, or they do not. If they overlap, then we have $|E_1 + \mathrm{fl}(E_2 - e_m)| < \max(|E_1|, |e_m|)$, which is too small to change s. If E_1 and e_m do not overlap, then $\mathrm{fl}(E_2 - e_m)$ can decrease E_1 by at most $\mathrm{ulp}(E_1)$, which again is too small to change s.] Thus, we only need to prove (9) implies (7). Since $e_m, E_1 \geq 0$ and Step 3(10)(a) yields (Figure 1),

$$|E_2| < \mathrm{ulp}(E_1) \tag{11}$$

we have, according to (9) and (11),

$$\mathrm{fl}(E_1 + E_2 + e_m) \geq E_1. \tag{12}$$

Hence, we obtain

$$|\mathrm{fl}(E_1 + E_2 + e_m)| = |\mathrm{fl}(E_1 + \mathrm{fl}(e_m + E_2))| < \mathrm{ulp}(s). \tag{13}$$

Case 2: if s is returned from Step 3(10)(c). In this case, we have $|E_1 + E_2| < \mathrm{ulp}(s)$ and $|e_m| < \mathrm{ulp}(s)$. Hence,

$$|\mathrm{fl}(E_1 + E_2 + e_m)| < 2\mathrm{ulp}(s). \tag{14}$$

Assume FastSum is called recursively and FastSum always returns at Step 3(10)(c). In each call of FastSum, we can obtain an s. Let s_j, for $j = 1, 2, \cdots, m$, denote the s in the j-th recursive call of FastSum, where s_1 has the largest absolute value. With (14), every two neighbors, say s_j and s_{j+1}, have at most one digit overlapping between their mantissas (see Figure 2). If using one floating-point number $\hat{s}$, whose mantissa is long enough, to represent the exact value of $\sum_{i=1}^{n} x[i]^{(0)}$, then the length of the mantissa of $\hat{s}$ is finite. Thus, m is finite. Furthermore, the last sum, s_m, returned by FastSum, is faithfully rounded, assuming the sums returned in Cases 1 and 3 are faithfully rounded. Therefore, the final result obtained by recursively summing s_j from $j = m$ to 1 is faithfully rounded.

This concludes the proof of the Theorem. $\square$

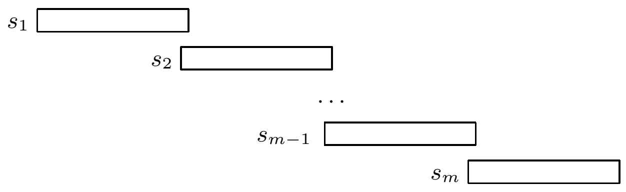


Fig. 2. Schematic illustration of the relationship between s_i, for $i = 1, 2, \cdots, m$.

2.3 Memory requirements, and discussion of Step 3(9)

The third case, if s is returned from Step 3(9), is not a part of our theorem, because we do not guarantee that s is faithfully rounded in this case. In a real-world implementation, the programmer would have a choice either of returning the "best-guess" for the sum that is currently stored in s, or of returning an error condition. However, we believe that it is possible to choose an L large enough so that s is never returned from Step 3(9). The smallest value of L required to avoid Step 3(9) is a function of machine arithmetic. We will first give a general discussion, and then details for IEEE754.

All the positive and negative numbers in x and q (including $e1$ and $e2$) can be regarded as two separate sequences, summing to s_p and s_n, respectively. We argue in the Appendix that the algorithm can reach a stable state after finite loops such that each AddTwo either does nothing, or simply swaps the two input addends. This occurs when no two numbers in x and q have overlapping mantissas, so that when applying AddTwo to any of them, the addend with greater magnitude becomes the sum, and the smaller one becomes the error. In this state, the values in the arrays x and q do not change. As argued in the Appendix, s_p (s_n) is the positive (negative) sum with a faithful rounding, and s is the whole sum with a faithful rounding. Furthermore, after finite loops, deflation causes each sequence to converge to being sorted by magnitude, with non-overlapping mantissas so that each element is the faithfully rounded sum of the elements before it, and none of the elements in the x and q arrays have overlapping mantissas with s. That is, our algorithm has in principle finished, in that

$$\mathrm{fl}\left(s + \sum_{i=1}^{n} x[i] + \sum_{j=1}^{c_q} q[j]\right) = s. \tag{15}$$

However, we cannot *detect* that we are finished because 3(10) is not satisfied. This occurs because e_m is a pessimistic upper bound on the sum of remaining terms in x and q, and it is *too* pessimistic when *count* in Step 3(4) is too big, which in turn occurs when there are too many non-zero elements in the arrays x and q. Thus it is possible that $\mathrm{fl}(s + e_m) \neq s$ while (15) is satisfied. We now argue that this can be avoided as long as the algorithm is allowed to converge to the non-overlapping state described in the Appendix.

As we argue in the Appendix, after a finite number of loops, no two pairs of floating-point numbers in x or q have overlapping mantissas. Once the algorithm has converged to this state, the value of *count* is limited by the number of non-overlapping floating-point numbers that can be represented by the machine. For example, in IEEE754 *double*, the length of exponent is 11 binary digits, and $t = 53$. Therefore, the maximum value of *count* in the converged state is $2^{11}/53 \approx 38.6$. When using IEEE754 single precision, the maximum is $2^8/24 \approx 10.7$. Furthermore, we have observed empirically (see below) that our algorithm converges to the the fully non-overlapping state in at most 48 *loop*s, requiring at most $L = 96$. In order for e_m to change s in this converged state of $\max(|s_p|, |s_n|) \approx \text{ulp}(s)$ and $e_m = count \cdot \text{ulp}(\max(|s_p|, |s_n|)$, we would require $e_m \approx \max(|s_p|, |s_n|)$, which would require $count \approx \beta^t$, which is typically far greater than 38.

We have set $L = 200$ in our algorithm, although we are almost certain that a value of 100 is sufficient.

3 Numerical results

In this section, we make comparisons among the following five methods: ordinary recursive summation, denoted by ORS; Sum3 ($K = 3$ for SumK) [6]; Algorithm 5 [9], denoted by Zhu05; AccSum [8]; and our new method FastSum. The floating-point arithmetic used here is IEEE754 *double*.

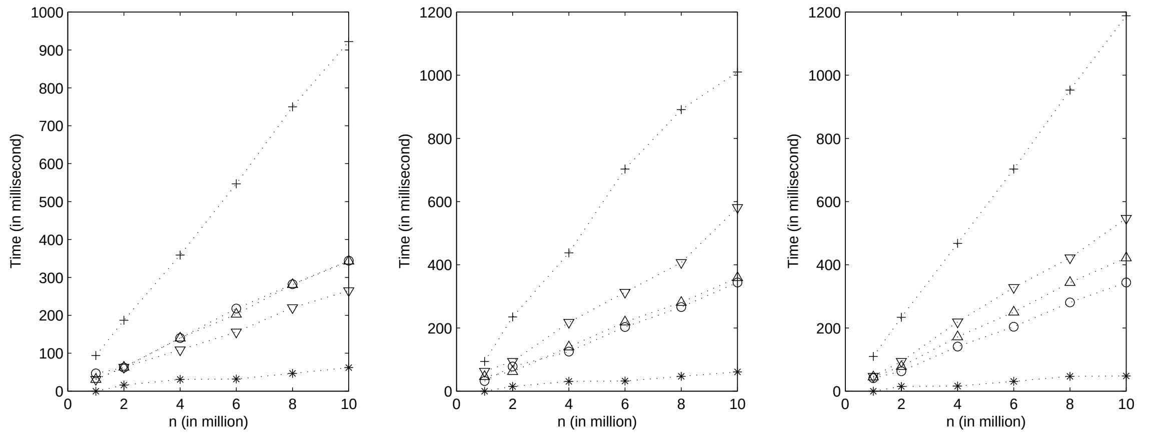
Three kinds of testing data are used in our numerical tests. The first one is the well-conditioned data, $R = 1$. Here, we randomly generate positive floating-point numbers. The second one is the ill-conditioned data, which are generated by the method mentioned in [1]: after randomly generating n floating-point numbers, the mean of the data (calculated using recursive summation) is subtracted from each datum. The condition number of this case is more than 10^9. The third one is the extremely ill-conditioned data, whose real sum is exactly zero, which are obtained by randomly generating a pair of floating-point numbers with the opposite signs, and then randomly disturbing the order of these n numbers.

All algorithms are implemented with Visual C++ 6.0, and run on a machine with a Pentium M 1.4GHz processor, 512MB memory, and Microsoft Windows XP Pro. In our numerical tests, ORS method never produces a correct sum, and Zhu05, AccSum and FastSum always yield results with faithful rounding. Sum3 fails when the third kind of data is used and the maximal binary exponent difference between two summands is more than 90.

The running times of five methods are shown in Figure 4, and the ratios of them to that of the ordinary recursive summation are listed in Table 1. The observation is that FastSum runs about three times faster than Zhu05. The running times of Sum3, AccSum and FastSum are similar, all about 6 times slower than ORS. We can see from Figure 4 that the running times of all the algorithms are essentially linear with n.

Data No.1	2	4	6	8	10 ($\times 10^6$)
Sum3	3.9	4.5	6.8	6.0	5.5
Zhu05	11.7	11.6	17.1	16.0	14.9
AccSum	4.0	3.5	4.9	4.7	4.3
FastSum	4.0	4.5	6.3	5.9	5.5
Data No.2	2	4	6	8	10 ($\times 10^6$)
Sum3	5.2	4.0	6.3	5.7	5.6
Zhu05	15.7	14.1	21.9	18.9	16.6
AccSum	6.3	7.0	9.8	8.7	9.5
FastSum	4.1	4.5	6.8	5.9	5.8
Data No.3	2	4	6	8	10 ($\times 10^6$)
Sum3	4.2	8.8	6.6	6.0	7.2
Zhu05	15.6	29.3	22.7	20.3	24.7
AccSum	6.3	13.7	10.5	9.0	11.4
FastSum	5.2	10.7	8.1	7.3	8.8

Table 1. Ratio of the running times of summation methods to that of the ORS.



(a) (b) (c)

Fig. 3. Running time of summation methods using different data sets: (a) Data No.1, well-conditioned; (b) Data No.2, ill-conditioned; (c) Data No.3, the real sum equals zero, where ORS is denoted by '$*$', Sum3 by '$\circ$', Zhu05 by '$+$', AccSum by '∇', and FastSum by '$\triangle$'

4 Discussion

In the above numerical tests, we observe that FastSum always generates a result within 2 loops (*loop* $= 2$, see Steps 1 and 3(1)). That is, Steps 2 and 3 each execute precisely once. Furthermore, all the results are returned from Step 3(10)(b). We now explain why we believe $L = 200$ will always give a return from Step 3(10)(b). We have never observed a return from Step 3(10)(c).

Our algorithm takes advantage of a small relative error of the ordinary recursive summation when all the summands have the same sign. Therefore, when heavy cancellation happens, i.e. the condition number is large, our algorithm can make quick cancellations between positive and negative numbers. Using the third kind of data set, $L = 200$, and then gradually increasing the binary exponent difference between two summands, we observe that when the maximal exponent difference is greater than 90, then Sum3 generates errors. When the difference is greater than 2000, then AccSum produces an overflow. However the result obtained by FastSum is always observed to be correct, and the biggest *loop* is 48. As for the first and second kinds of data, the value of *loop* is always 2 even if we increase the exponent difference, since the condition number does not change much. Thus, we need a large L only when cancellation dominates.

In the case that the exact sum is zero (as in our third kind of data), our algorithm must cancel all the significant digits to produce a correct result. Assume we cancel m digits with each loop, and the maximal exponent difference between arbitrary two numbers is d. Then, our algorithm requires about $\lceil d/m \rceil$ loops. Since in IEEE754 *double*, $d = 2046$, if we let $m = t = 53$, we have $\lceil d/t \rceil = 39$, assuming the algorithm can cancel 53 bits per *loop*, which is optimistic. Since our maximum *loop* is 48, we see that our algorithm cancels on average perhaps about 44 bits per *loop*. Thus, choosing $L \approx 4d/t$ should be enough. In our test, we let L be 200, which allows Step 3 to loop at most 100 times. In practice, the case $d \gg t$ is probably not realistic, since either $s \ll \mathrm{ulp}(M)$, where

$$M = \max_{1 \le i \le n} \left(\left| x[i]^{(0)} \right| \right),$$

and so s probably does not represent a meaningful quantity; or $s \gg S$, where

$$S = \min_{1 \le i \le n} (\left| x[i]^{(0)} \right|).$$

Therefore, to save the space, we can make the size of q dynamically increasing from a small value (for instance, $L = 6$), since we believe that there exists an L which can let the algorithm return a faithfully rounding result. If it is known in advance that *loop* will be big (i.e., the sum is badly ill-conditioned and the maximum exponent difference is large), we note that each *loop* obliterates at least two values in the x array, so that the number of non-zero values is always decreasing. Thus, it may be advantageous to alternate between two copies of the x array, say x_0 and x_1, copying only non-zero values from one to the other, and switching which one to use on the next *loop*, thus never having to deal with 0 values in Step 3(2). A similar approach can be used for Step 3(3). We did not implement this, because $loop > 2$ occurs only rarely.

Another question regards whether the many branches in our algorithm affect its efficiency. We tested the two AddTwo algorithms TwoSum [6] and Algorithm2$'$ [9]. TwoSum is proposed by Knuth [5] which contains six floating-point addition operations without any branches. Algorithm2$'$ is presented by Dekker [2] which has only three floating-point addition operations, but a branch is needed. Another aspect of efficiency is whether AddTwo requires a function call, or if the instructions are executed “inline”. Table 2 gives the timings of FastSum under these four options. The environment is a Pentium M 1.4GHz processor under Windows XP Professional, compiled with Visual C++ 6.0. The compiler optimization option is “maximize speed”, which is the default option for “Win32 Release” in Visual C++ 6.0. We observed that using Dekker's algorithm is always faster than using Knuth's algorithm. Thus, in our environment, empirical observations suggest that branches are not important. Although we find this surprising, we hypothesize that perhaps the Pentium's hardware branch prediction algorithm and optimal pipelining might reduce the cost of branches when the number of instructions inside the if/else construct is small.

	with function calls	without function calls
Using Dekker's algorithm	437	375
Using Knuth's algorithm	469	391

Table 2. Running times (in milliseconds) of FastSum with four options.

Finally, we observe that the running time of FastSum is linear with n. In our numerical tests, we only give the results when n is huge. But we can compute its running time for another n from the existed results directly. For example, from Figure 4(c), the running time of FastSum when $n = 10,000,000$ is about 410 milliseconds. After testing, we found that the running time for FastSum ($n = 1,000$) running 10,000 times is also about 410 milliseconds, as it is for $n = 100$ running 100,000 times, etc.

5 Conclusions

In this paper, we present a fast and accurate algorithm, FastSum. This algorithm improves upon [9], by running three times faster. In general cases, even if the condition number is about 10^9, our algorithm returns a faithfully rounded result with two loops ($loop = 2$), and is never observed to require more than 48 loops, even in the worst-case of very unrealistic, contrived input data.

We depend on the existence of AddTwo, which is a black box for calculating the floating-point sum and the error of two arbitrary floating-point numbers, and on this condition our algorithm works independently of the base β of the floating-point arithmetic. Furthermore, we have proved that its accuracy does not depend on n or the condition number, although its running time may depend on n and the condition number in rare cases. We have also observed, and argued informally, that beyond the input array itself, the additional memory requirement is constant, although we have not formally proven it.

References

1. I. J. Anderson. A distillation algorithm for floating-point summation. *SIAM Journal on Scientific Computing*, 20(5):1797–1806, 1999.
2. T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.
3. N. J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993.
4. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM. Philadelphia, second edition, 2002.
5. D. E. Knuth. *The art of computer programming*, volume 2/Seminumerical algorithms. Addison-Wesley, third edition, 1998.
6. T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26:1955–1988, 2005.
7. D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, Mathematics Department, University of California, Berkeley, CA, 1992.
8. S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation. Technical report 05.12, Faculty for Information and Communication Sciences, Hamburg University of Technology, November 2005.
9. Y.-K. Zhu, J.-H. Yong, and G.-Q. Zheng. A new distillation algorithm for floating-point summation. *SIAM Journal on Scientific Computing*, 26:2066–2078, 2005.

Appendix

Lemma A

In this section, we will show that

> *Claim 1*: After finite loops FastSum can reach a stable status such that each AddTwo either does nothing, or simply swaps the two input addends, so that all the positive and negative elements, adding to s_p and s_n respectively, and thus the total sum s, are constant.

Our claim is based on the following Lemma.

```
ALGORITHM: SumA($w, n_w$)
    Input: $w$, the array of the given floating-point summands;
           $n_w$, the number of summands;

1. while (any two elements overlap)
   (1) for $i \leftarrow 2$ to $n_w$
       (a) $\{w[i], w[i-1]\} = \text{AddTwo}(w[i], w[i-1])$
2. END
```

Lemma A. *Algorithm SumA halts, at which time (i) no term overlaps any other, and (ii) each term w_i is a faithful sum of the terms before it.*

Outline of proof. We omit the full proof (a) due to space limitations, and (b) because we believe this case is never satisfied, although we have not been able to prove that it is never satisfied. However, we will give a flavour of the proof of Lemma A.

The proof of part (i) is a direct consequence of the while statement in the algorithm; (ii) can be proved by contradiction. Let us imagine that the value of each $w[i]$ represents mass. The fundamental observation is that the inner loop transfers mass towards higher-numbered elements (i.e., towards $w[n_w]$), while residuals are pushed towards lower-numbered elements (i.e., towards $w[1]$). To see this, let us focus our attention on one particular element of the array, say $w[k]$, $1 < k < n$. On each execution of the outer loop, the inner loop touches $w[k]$ twice, once when $i = k$ and once when $i = k + 1$. Immediately after the $i = k$ iteration, the mantissas of elements $w[k]$ and $w[k - 1]$ do not overlap, by the definition of AddTwo. Effectively, if $w[k - 1]$ and $w[k]$ *did* overlap before the AddTwo, then the AddTwo transfers some mass from $w[k - 1]$ to $w[k]$, shrinking $w[k - 1]$ so that it does not overlap with $w[k]$. Similarly, on the $i = k + 1$ iteration, $w[k]$ might shrink as some of its mass is transferred into $w[k + 1]$—thus shifting $w[k]$ down so that it again overlaps with $w[k - 1]$. The net effect (and this is where the formal proof gets messy) is to transfer mass towards higher indices, and to reduce the amount of overlap between adjacent elements of the array. This process eventually converges, in finite iterations of the outer loop, to the point that no element overlaps any other. At this point some of the lower-indexed elements may be zero, and in fact some of them *must* be zero if n_w is greater than the greatest number of possible non-overlapping mantissas in the machine (38 in the case of IEEE754 double precision). Furthermore, by the definition of AddTwo, no mass is ever discarded, so that the total sum of material in the array never changes. But once the values converge so that none of the mantissas overlap each other, and they are sorted by increasing magnitude. Furthermore, the sum of the mass in elements 1 through $i - 1$ can have no effect on the value of element i, since every element is smaller than the `ULP` of the next. In other words, at convergence, each element $w[i]$ represents a faithful sum of the elements before it. This completes the outline of the proof of Lemma A. □

Application of Lemma A

Note that after *loop* iterations of the main loop of our main algorithm, $x[1], x[2], \cdots, x[loop]$ are all identically zero. Thus, after at most n iterations of the loop, all elements of x are zero. After this point, each iteration of the main loop similarly obliterates one value in the q array, so that $q[loop - n] = 0$ as well.

We apply Lemma A independently to the values s_p and s_n, to show that equation (15) eventually holds, after finite iterations of the main loop on Step 3. To apply Lemma A, we first concatenate the arrays x and q together into one array z such that $z[1] = x[1], z[2] = x[2], \cdots, z[n] = x[n], z[n + 1] = q[1], z[n + 2] = q[2], \cdots$.

In order for L to be unbounded, c_q must grow without bound, incrementing by one for each iteration of the outer loop. However, as we have shown in the proof of the theorem, the value of *count*, and thus the total number of non-zero elements in the arrays x and q is never greater than n. Thus, the string of non-zero elements marches along the z array, with no non-zero elements below index *loop*, and no non-zero elements beyond index c_q, where $c_q - loop \leq count \leq n$. Thus, we apply Lemma A to the elements $z[loop + 1]$ though $z[loop + c_q]$, which, finally, demonstrates that (15) is true after finite loops. So, s is the sum with a faithful rounding. Since the sum and the number of summands are finite, after enough cancellations between s_p and s_n, it can be satisfied that $\mathrm{fl}(s_p + s_n) = s_p$ or $\mathrm{fl}(s_p + s_n) = s_n$, and both the positive sequence and the negative sequence satisfy the lemma. Claim 1 follows as a corollary.

We emphasize again that we believe that this case never occurs as long as L is large enough, although we have not yet been able to prove it.

Implementation of Float-Float Operators on Graphics Hardware

Guillaume Da Graça, David Defour

Dali, LP2A, Université de Perpignan,
52 Avenue Paul Alduy,
66860 Perpignan Cedex, France

Abstract. The Graphic Processing Unit (GPU) has evolved into a powerful and flexible processor. The latest graphic processors provide fully programmable vertex and pixel processing units that support vector operations up to single floating-point precision. This computational power is now being used for general-purpose computations. However, some applications require higher precision than single precision. This paper describes the emulation of a 44-bit floating-point number format and its corresponding operations. An implementation is presented along with performance and accuracy results.

1 Introduction

There is significant interest in using graphics processing units (GPUs) for general purpose programming. These GPUs have an explicitly parallel programming model and deliver much higher performance for some floating-point workloads when compared to CPUs. This explains the growing concern in using a graphic processor as a stream processor for executing highly parallel applications.

1.1 The graphics pipeline

Data processed by the GPU are mainly pixel, geometric objects and elements that create the final picture in the frame buffer. These objects require an intensive computation before getting the final image. This computation is done within the "Graphics Hardware Pipeline". The pipeline contains several steps in which the 3D application sends a sequence of vertices to the GPU that are batched into geometric primitives (polygons, lines, points). These vertices are processed by the programmable vertex processor that has the ability to perform mathematical operations. Then the resulting primitives are sent to the programmable fragment processor. Fragment processors require the same mathematical operation as the vertex processors, plus some texturing operations. A representation of this pipeline is shown in figure 1.

The computational workhorse of the GPU is located within the 2 programmable processors: vertex processors and fragment processors. The amount of these processors embedded in GPUs has greatly increased over the years; for example the latest Nvidia 7800GTX chip integrates 8 vertex shaders and 24 pixel shaders. In this chip, each vertex shader is made up of 1 multiply and accumulate (MAD) unit and 1 special function unit that can compute log, exp, sin, cos. The implementation of a similar unit is detailed in [18]. Each pixel shader of the 7800GTX consists of 2 consecutive MADs, therefore the 7800GTX is able to execute 56 MAD which correspond to 112 floating-point operations in single precision per clock cycle at a peak rate.

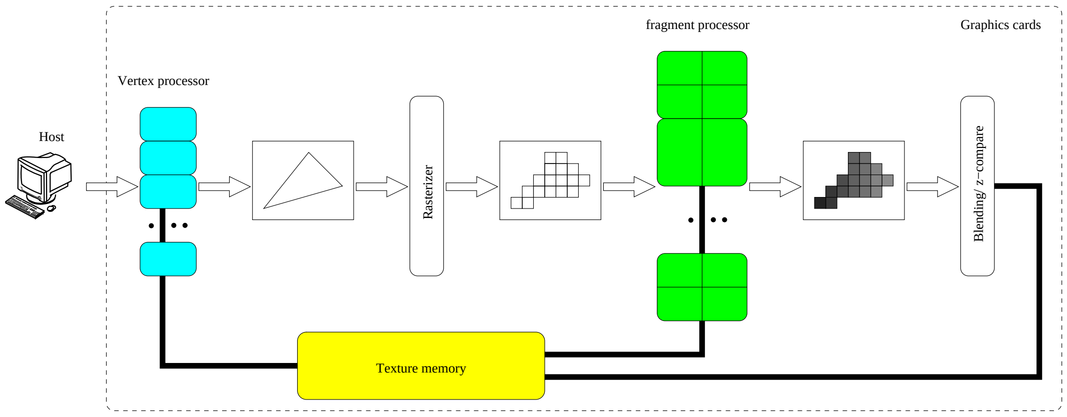


Fig. 1. This figure illustrates the current graphics pipeline found in recent PC graphics cards.

1.2 Representation formats available in GPUs

Vertex and pixel shaders were originally composed of fixed-point operators that have evolved into partial support of the IEEE-754 single precision floating-point format. For example, the Nvidia GeForce 6 series offers a 32-bit format similar to single precision. The other main GPU manufacturer, ATI, integrated a 32-bit floating-point arithmetic required by shader version 3.0 in their latest chips, the X1k series. Older ATI hardware performed floating-point operations on a 24-bit format in spite of the fact that they stored values in the IEEE standard 32-bit format as described in table 1.

Format name	Sign	Exponent	Mantissa	Support for special values (NaN, Inf)
Nvidia 16-bit	1	5	10	Yes
Nvidia 32-bit	1	8	23+1	Yes
ATI 16-bit	1	5	10	No
ATI 24-bit	1	7	16	No
ATI 32-bit	1	8	23+1	? / *not tested*

Table 1. Floating-point format currently supported by the Nvidia and the ATI chips.

In addition to these formats, current GPUs support other data types that are of lower precision. Therefore, applications where accuracy is paramount are not well suited for a GPU execution due to the lack of the double precision format, the non uniformity within the floating-point format and the non-respect of the IEEE-754 requirement (such as rounding or denormal number which are typically flushed to zero [6]).

The purpose of this article is to propose a software solution to the limitation of precision in floating-point operations and storage. This solution consists of an implementation of a *float-float* format which doubles the hardware accuracy. This corresponds to a 44-bit precision on Nvidia architecture.

1.3 Outline of this paper

Based on the above clarification, hence section 2 presents similar work related to multiprecision operators. Section 3 describes the floating-point arithmetic available in current GPU. Section 4 proposes a representation and the algorithms for the basic multiprecision operations. Section

5 describes our initial implementation used to conduct tests and comparisons which results are discussed in section 6.

2 Related work

Many modern processors obey the IEEE-754 [15] standard for floating-point arithmetic, which defines the single and double precision format. For some applications, however, the precision provided by the hardware operators does not suffice. These applications include large scale simulation, number theory and multi-pass algorithm like shading, lighting [21].

Applications encountering accuracy problems are commonly executed on a CPU. As a consequences, most of the research was done to develop a *multiprecision* format on the CPU (a representation format with a precision higher than the one available in the hardware).

2.1 CPU related work

Many software libraries were proposed to address the precision issue in hardware limitation. These libraries emulate arithmetic operators with higher precision than the one provided by the hardware. They either use integer units or floating-point units, depending on the internal representation of their number.

Libraries based on an integer representation All the libraries in this category, internally represent multiprecision numbers as an array of integers, which are machine numbers (usually 32-bit or 64-bit) to store the significant of the multiprecision numbers. It is the case for GMP [9], on top of which several other libraries are built (see MPFR [2]). These libraries allow the user to dynamically set the precision of the operation during the execution of the program. However some other libraries [1, 7] set the precision at compilation time to get higher performance.

Libraries based on a floating-point representation The actual trend of CPUs is to have highly optimized floating-point operators. Some libraries, such as the MPFUN [3], exploit these floating-point operators by using an array of floating-point numbers.

Other libraries represent multiprecision numbers as the unevaluated sum of several double-precision FP numbers such as Briggs' double-double [4], Bailey's quad-doubles [13] and Priest's floating-point expansions [19]. This representation format is based on the IEEE-754 features that lead to simple algorithms for arithmetic operators. However this format is confined to low precision (2 to 3 floating-point numbers) as the complexity of algorithms increases quadratically with the precision.

2.2 GPU related work

The available precision provided through the GPU's graphical pipeline is limited; for example the color channel is usually represented with a 8-bit number. Before the introduction of the shader 3.0 that requires support of 32-bit floating-point numbers, developers and researchers that were facing accuracy problems developed software solutions to extend the hardware precision.

For example, Strzodka [21] proposed a 16-bit fixed-point representation and operation out of the 8-bit fixed-point format. In his work, two 8-bit numbers were used to emulate 16-bit.

The author claimed that operators in his representation format were only 50% slower than normal operators, however no measured timings were provided. Strzodka recently implemented a solvers for Finite Element simulations in double precision on GPU [11], nevertheless double precision computation were sent to the CPU. This method involves time consuming memory transfer.

3 Floating-point arithmetic on GPUs

Floating-point computations on GPUs are often called into question. Current GPUs do not strictly conform to the IEEE-754 floating-point standard. This produces differences between the same computation performed on the GPU and the CPU, and among GPUs themselves. Floating-point computation details vary with GPU models and they are kept secret by GPU manufacturers.

Recently, one tool has been developed to understand some of the details of the floating-point arithmetic for a given GPU [14]. We executed this tool on GPUs, which is an adaptation of Paranoia, and got some resulting errors reported in table 2. This tools is only available as an executable with poor documentation so these data should be taken with caution and real bounds may be larger. We will assume in the rest of this paper that these bounds hold for all possible inputs.

Operation	Exact rounding	Chopped	R300	NV35
Addition	[-0.5, 0.5]	(-1, 0]	[-1.0, 0.0]	[-1.0, 0.0]
Subtraction	[-0.5, 0.5]	(-1, 1)	[-1.0, 1.0]	[-0.75, 0.75]
Multiplication	[-0.5, 0.5]	(-1, 0]	[-0.989, 0.125]	[-0.782, 0.625]
Division	[-0.5, 0.5]	(-1, 0]	[-2.869, 0.094]	[-1.199, 1.375]

Table 2. Floating-point error in ulp from the execution of the paranoia Test [14]

Table 2 shows us that the addition is truncated after the last bit on both ATI R300 and Nvidia NV35. This software tells us that the subtraction is done with an extra bit (guard bit) on Nvidia processors and not on ATI. This property is very important for numerical algorithms as we will see later on in this paper. The error of the multiplication is strictly lower than one ulp (faithfully rounded) on both the ATI and the Nvidia. Because GPUs do not provide a division instruction, every division is performed as a reciprocal followed by a multiplication; thereby the floating-point error for the division incurs double floating-point errors.

4 Proposed format

The proposed format is an adaptation of the double-double format described in [4]. For our algorithm we chose to represent multiprecision numbers as the unevaluated sum of 2 floating-point numbers handled in hardware. The type of hardware representation used is described in Table 1.

In the core of the GPU, the graphical pipeline is made up of several computational units. These processing units are not designed to efficiently perform tests and comparisons, therefore whenever it is possible, we should avoid tests even at the expense of extra computations. In addition, software that does not use branches remains compatible with older GPU. In our case, two versions of Add12 algorithms exist [20]; one with one test and another one, that should be preferred, with 3 extra floating-point operations.

4.1 Mathematical background

In this section we present some basic properties and algorithms of the IEEE floating-point arithmetic used in our format. In this paper we assume that GPUs have a guard bit for the addition/subtraction with a truncation as rounding mode as it seems to be the case with latest Nvidia chips. We will consider single precision number. The multiplication will behave as observed in section 3. This assumption conforms to the actual trends follow by the GPU and the shader model 3.0. For any mathematical operator $+, -, *, /$, we use $\oplus, \ominus, \otimes, \oslash$ to represent the hardware operator that may involve a rounding error.

Theorem 1 (Sterbenz lemma ([12] Th. 11)). *If subtraction is performed with a guard digit, and $y/2 \leq x \leq 2y$, then $x \ominus y$ is computed exactly.*

Theorem 2 (Add12 theorem (Knuth [16])). *Let a and b be normalized floating-point numbers. The following algorithm computes $s = a \oplus b$ and $r = (a+b) - s$ such that $s + r = a + b$ exactly, provided that no exponent overflow or underflow occurs.*

```
Add12(a, b)
$s = a \oplus b$
$v = s \ominus a$
$r = (a \ominus (s \ominus v)) \oplus (b \ominus v)$
return (s, r)
```

Proof. The proof of correctness of the Add12 algorithm, in an environment with a correctly rounded arithmetic, is described in Knuth [16] or Shewchuk [20] articles. However, by examining these proofs, one can observe that they are based on Sterbenz lemma. This lemma only requires a guard bit to be true, therefore the Add12 theorem is true on Nvidia hardware. □

Theorem 3 (Split theorem (Dekker [8])). *Let a be p-bit floating-point number, where $p \geq 3$. Choose a splitting point s such that $p/2 \leq s \leq p-1$. Then the following algorithm will produce a $(p-s)$-bit value a_{hi} and a non-overlapping (s)-bit value a_{lo} such that $|a_{hi}| \geq |a_{lo}|$ and $a = a_{hi} + a_{lo}$.*

```
SPLIT( a )
1 $c = (2^s \oplus 1) \otimes a$
2 $a_{big} = c \ominus a$
3 $a_{hi} = c \ominus a_{big}$
4 $a_{lo} = a \ominus a_{hi}$
5 return $(a_{hi}, a_{lo})$
```

Proof. This proof is an adaptation of the proof from [20] to fit the condition observed on GPUs. Line 1 is equivalent to computing $2^s a \oplus a$, because multiplying by a power of two only changes its exponent. The addition is subject to rounding, so we have $c = 2^s a + a + err(2^s a \oplus a)$. Line 2 is subject to rounding, so $a_{big} = 2^s a + err(2^s a \oplus a) + err(c \ominus a)$. Both $|err(2^s a \oplus a)|$ and $|err(c \ominus a)|$ are bounded by $ulp(c)$, so the exponent of a_{big} can only be larger than that of $2^s a$ if every bit of the significand of a is nonzero except the last two bits. By manually checking the behavior of SPLIT in these 4 cases, one can verify that the exponent of a_{big} is never larger than that of $2^s a$. Then $|err(c \ominus a)| \leq ulp(2^s a)$, and so the error term $err(c \ominus a)$ is expressible in s bits.

By Sterbenz lemma line 3 and 4 are calculated exactly. It follows that $a_{hi} = a - err(c \ominus a)$ and $a_{lo} = err(c \ominus a)$; the latter is expressible in s bits. Either a_{hi} has the same exponent as a either a_{hi} has an exponent one greater than that of a and in both case a_{hi} is expressible in $p - s$ bits. □

Theorem 4 (Mul12 theorem (Dekker [8])). *Let a and b be normalized floating-point numbers. The following algorithm produces two floating point numbers x and y as results such that $a \cdot b = x + y$, where x is an approximation to $a \cdot b$ and y represents the roundoff error in the calculation of x.*

```
Mul12(a,b)
1 x = a ⊗ b
2 (a_hi , a_lo) = SPLIT (a)
3 (b_hi , b_lo) = SPLIT (b)
4 err1 = x ⊖ (a_hi ⊗ b_hi)
5 err2 = err1 ⊖ (a_lo ⊗ b_hi)
6 err3 = err2 ⊖ (a_hi ⊗ b_lo)
7 y = (a_lo ⊗ b_lo) ⊖ err3
8 return (x,y)
```

Proof. Line 1 computes $x = ab + err(a \otimes b)$ with $err(a \otimes b)$ the rounding error of the multiplication. One can noticed that all the other multiplications and subtractions are exact and compute $y = -err(a \otimes b)$. □

Theorem 5 (Add22 theorem). *Let $ah + al$ and $bh + bl$ be the float-float arguments of the following algorithm:*

```
Add22(ah, al, bh, bl)
1 r = ah ⊕ bh
2 if |ah| ≥ |bh| then
3     s = (((ah ⊖ r) ⊕ bh) ⊕ bl) ⊕ al
4 else
5     s = (((bh ⊖ r) ⊕ ah) ⊕ al) ⊕ bl
6 (rh, rl) = Add12(r, s)
7 return (rh, rl)
```

The two floating-point numbers rh and rl returned by the algorithm verify

$$rh + rl = (ah + al) + (bh + bl) + \delta$$

Where δ is bounded as follows:

$$\delta \leq max(2^{-24} \cdot |al + bl|, 2^{-44} \cdot |ah + al + bh + bl|)$$

Theorem 6 (Mul22 theorem). *Let $ah + bl$ and $bh + bl$ be the float-float arguments of the following algorithm:*

```
Mul22(ah, al, bh, bl)
1 (t1, t2) = Mul12(ah, bh)
2 t3 = ((ah ⊗ bl) ⊕ (al ⊗ bh)) ⊕ t2
3 (rh, rl) = Add12(t1,t3)
4 return (rh, rl)
```

The result $rh + rl$ returned by the algorithm verify

$$rh + rl = ((ah + al) * (bh + bl)) * (1 + \epsilon)$$

Where ϵ is bounded as follows:

$$|\epsilon| \leq 2^{-44}$$

Proof. The detailed proof of the Add22 and the Mul22 theorem were proposed by Lauter in [17] for the particular cases of double-double format on an IEEE compliant architecture. The proof of these 2 theorems with GPU conditions (single precision, faithful rounding and guard bit) is very similar and is therefore not detailed here for the sake of brevity. □

5 Implementation

We developed a Brook [5] implementation of the float-float format and of Add12, Add22, Split, Mul12, Mul22 algorithms. Brook is a high level programming language designed for general purpose programming on GPUs. This language allows us to test our algorithms with ease over various systems, drivers and graphics hardware with minor modifications.

During our implementation, we observed that the DirectX version generated by Brook were performing forbidden floating-point optimization. These floating-point optimizations were not noticed with the OpenGL version. For example, the sequence of operations that compute the rounding error $r = ((a \oplus b) \ominus a)$ was replaced by $r = b$. To overcome this problem, we had to apply hand correction on the fragment program generated by Brook.

6 Results and performance

It is quite difficult to compare the performances of GPU and CPU operators. CPUs already have data stored in the memory hierarchy whereas GPUs have to download data from main memory to its local memory before processing it. To make fair comparisons, we compared float-float algorithms to basic single precision operations (addition, multiplication, multiply and add) on a CPU and on a GPU. For clarity we normalized results to the time of 4096 additions. For each version we tested different sizes of data set. Tests were done on a Nvidia 7800GTX graphics card with 256 MB with a core frequency at 430 Mhz and on a Pentium IV HT at 3.2 Ghz.

We have not reported the difference of execution time between CPUs and GPUs because such comparison is meaningless. However to give an idea, we measured that sending data to the GPU, executing the 4096 additions and getting back the results on the CPU correspond to 100 times the execution time of the same 4096 additions on the CPU. This overhead mainly comes from the use of the bus of the system to send and to get back data. Therefore GPU will be faster than CPU if many operations will be done on the same large set of data.

The difference of time between small and large data set is higher for the CPU than for the GPU. This difference is of 25 for GPU and 3000 for CPU. This means that GPUs are more efficient at performing the same operation over a large set of data. The Add22 times on CPU is much higher than other operations. An interpretation could be that the test in the Add22 algorithm is time consuming compared to normal operations as it breaks the execution pipeline.

Size	Add	Mull	Mad	Add12	Mul12	Add22	Mul22
4096	1.00	0.97	1.00	1.09	1.57	1.55	1.54
16384	1.11	1.11	1.15	1.20	1.87	1.73	2.02
65536	1.55	1.58	1.69	1.64	2.09	2.87	2.94
262144	3.55	3.40	3.44	3.74	3.99	7.15	7.47
1048576	10.64	10.74	10.75	10.79	14.64	23.92	24.64

Table 3. Timing comparison of float-float operators executed on the GPU. The time is normalized on the single addition of 4096 data.

Size	Add	Mull	Mad	Add12	Mul12	Add22	Mul22
4096	1.00	0.98	1.35	1.52	2.86	11.71	4.12
16384	3.88	3.88	3.46	6.04	17.86	47.93	17.62
65536	17.13	16.20	17.67	28.35	49.14	192.10	69.33
262144	68.77	66.68	77.10	100.10	187.49	760.65	272.13
1048576	269.49	267.88	312.45	419.84	1027.62	3083.74	1091.59

Table 4. Timing comparison of float-float operators executed on the CPU. The time is normalized on the single addition of 4096 data.

We observe that the execution time of the addition, the multiplication, the multiply and accumulate and the Add12 is about the same as the GPU one. The algorithms Add22 and Mul22 cost twice as much as basic operations. This proves that GPU drivers are very efficient at merging operations of different execution loops. This also signifies that the cost of these operations could be higher when used in a real program.

6.1 Accuracy

We ran our algorithms on 2^{24} randomly generated test vectors and we collected the maximum observed error with the help of MPFR [2]. For these tests, we excluded denormal input numbers as they are not supported by the targeted hardware. The number of bits of accuracy per operation is presented in table 5.

Operation	Error max
Add12	-48.0
Mul12	(exact)
Add22	-33.7
Mul22	-45.0

Table 5. Number of correct bits per operation on an Nvidia 7800 GTX chip of our GPU float-float implementation

The first observation we can make is that the reported accuracy is different from the theoretical one. We proved in section 4 that if GPUs have a guard bit then Add12 will be exact. Our initial tests show us that GPUs behave as if they have a guard bit. However in a very special case the error is higher than expected. This happens when two floating point numbers of opposite signs are summed up together and when their mantissa are not overlapping in a certain way. For example, it happened with $a = 1.0$ and $b = -(2^{24} - 2^{48})$. Further investigations have to be done to locate and correct the problem. This problem is also the source of the bad accuracy result of the Add22 algorithm.

7 Conclusion and future work

In our work, we have described a general framework for the implementation of software emulation of floating-point numbers with 44 bits of accuracy. The implementation is based on Brook and allows simple and efficient addition, multiplication and storage of floating-point number. The representation range of this format is similar to single precision. These high precision operations naturally require more texture memory and computing time. However, they proved to remain fast enough to be used in precise sensitive parts of real-time multipass algorithms.

During our tests, we noticed that Brook was well suited for fast prototyping of functions; however, as with every high level languages, we were unable to have fine control over GPUs instructions. In particular, it was not possible to control how and when data were stored, transferred and used within the GPU. Therefore, we are currently working on translating these functions in a lower level graphic langage (OpenGL and Cg [10]). We hope that it will lead to an improvement in performance. We are also investigating to set a solution to the accuracy problem described in section 6.1. Using float-float representation number in compensated algorithms has been shown to be more efficient in term of performance for comparable accuracy. Adapting compensated algorithm to GPU is part of our future investigation.

References

1. IBM accurate portable mathematical library.
2. MPFR, the Multiprecision Precision Floating-Point Reliable library. `http://www.mpfr.org/`.
3. D. H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, 1995.
4. K. Briggs. The doubledouble library, 1998. `http://members.lycos.co.uk/keithmbriggs/doubledouble.html`.
5. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of SIGGRAPH 2004*, pages 777 – 786, August 2004.
6. Cem Cebenoyan. Floating point specials on the GPU. Technical report, Nvidia, February 2005.
7. D. Defour and F. de Dinechin. Software carry-save: A case study for instruction-level parallelism. In *7th conference on parallel computing technologies*, Nizhny-Novgorod, September 2003.
8. T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
9. Torbjörn Granlund et al. GNU multiple precision arithmetic library. `http://swox.com/gmp/`.
10. Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
11. Dominik Goddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision fem simulations with GPUs. In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, September 2005.
12. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar 1991.
13. Y. Hida, X. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, Jun 2001.
14. Karl Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, page C8, August 2004.
15. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.
16. D. Knuth. *The Art of Computer Programming*, volume 2, ”Seminumerical Algorithms”. Addison Wesley, Reading, MA, third edition, 1998.
17. Christoph Quirin Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, École Normale Supérieure de Lyon, September 2005.

18. Stuart F. Oberman and Michael Siu. A high-performance area-efficient multifunction interpolator. In Koren and Kornerup, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (Cap Cod, USA)*, pages 272–279, Los Alamitos, CA, July 2005. IEEE Computer Society Press.
19. D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.
20. Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
21. R. Strzodka. Virtual 16 bit precise operations on rgba8 textures. In *Proceedings of Vision, Modeling, and Visualization*, pages 171–178, 2002.

An Ordinary Differential Equation Defined by a Computable Function whose Maximal Interval of Existence is Non-Computable

Daniel S. Graça[1,2], Ning Zhong[3], and Jorge Buescu[4]

[1] DM/FCT, Universidade do Algarve, C. Gambelas, 8005-139 Faro, Portugal
[2] CLC, DM/IST, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal
[3] DMS, University of Cincinnati, Cincinnati, OH 45221-0025, U.S.A.
[4] CAMGSD, DM/IST, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal

Abstract. Let $(\alpha, \beta) \subset \mathbb{R}$ denote the maximal interval of existence of solution for the initial-value problem

$$\begin{cases} \frac{dx}{dt} = f(t,x), & f : E \to \mathbb{R}^m, E \text{ is an open subset of } \mathbb{R}^{m+1} \\ x(t_0) = x_0, & \text{with } (t_0, x_0) \in E. \end{cases}$$

We show that (α, β) is r.e. (recursively enumerable) open and the solution $x(t)$ defined on (α, β) is computable, provided that (a) f is computable and effectively locally Lipschitz, and (b) (t_0, x_0) is a computable point. We also prove that this result is the best in the sense that, for some initial-value problems satisfying (a) and (b), their maximal intervals of existence are non-recursive.

1 Introduction

Differential equations are fundamental in modelling physical processes, including nonlinear systems of ordinary differential equations (ODEs for short) $\dot{x} = f(t,x)$, where $f : E \to \mathbb{R}^n$, E is an open subset of $\mathbb{R}^{n+1}$, $x = x(t)$ is a function of t, and $\dot{x}$ denotes the derivative of x with respect to t. The well-posedness of such systems has long been established in the fundamental existence-uniqueness theorem [CL55], which states that, under appropriate regularity conditions for f, the initial value problem

$$\dot{x} = f(t, x(t)), \; x(t_0) = x_0 \tag{1}$$

(for short, we sometimes write $f(t,x)$ instead of $f(t,x(t))$) has a unique solution $x(t)$ defined on a maximal interval of existence $(\alpha, \beta) \subset \mathbb{R}$. In general, however, it is not possible to solve the above nonlinear system analytically with an explicit solution formula. Many of those nonlinear systems can only be solved numerically on computers and indeed computers are playing an ever larger role in studying differential equations. Numerical methods are usually tailor-made for individual problems and often depend on certain assumptions, for example, the existence of some time interval where the solution is defined. This requirement is crucial but in general hard to verify. In practice, there are several approaches to deal with this problem. One possible approach is to take some insight from the physical counterpart of the ODE. For example, if an ODE is intended to model the orbit of the Earth around the Sun, then we may assume that the solution is defined for $t \in [t_0, \infty)$. However, from a mathematical point of view, this certainly is not a very satisfactory solution.

The satisfactory solution of course is to have an "automated method" that determines the maximal interval (α, β) and computes the solution on (α, β) from the data defining the IVP (1). Thus, it becomes useful to know whether it is possible to derive such "automated method"?

In this note, we present a negative answer to the question. We show that the maximal interval where the solution of the IVP (1) is defined may not be recursive, even when the function f is computable and of class C^∞. In such circumstances, there is no algorithm to decide whether or not $[t_0, t]$ is contained in (α, β) from the information that $t_0 \in (\alpha, \beta)$ for arbitrary $t \geq t_0$. The undecidability indicates that the limit behavior of the IVP (1) may not be determined by "general numerical recipes". In other words, such undecidability suggests possible limitations concerning numerical methods for solving ODEs.

There are other noncomputability results related to the initial value problems of differential equations. For example, Pour-El and Richards [PER79] showed that the IVP (1) defined with computable data may have noncomputable solutions. In [PER81], [WZ02] it is shown that there is a three-dimensional wave equation, defined with computable data, such that the unique solution is nowhere computable. However, in these examples, noncomputability is not "genuine" in the sense that the problems in the study are ill-posed: either the solution is not unique or the solution is not stable. In other words, ill-posedness generated noncomputability in those examples. In contrast, all IVPs studied in this note are classically well-posed. For reference we also mention the existence of other results about computability of ODEs that can be found in [Abe70], [Abe71], [BB85], [Ko91], [Ruo96].

The computational model used in this paper is the Turing machine-based "bit" model [PER89], [Ko91], [Wei00]. This approach is based on the classical theory of computability, where an approximation of the output with arbitrary precision is computed from a suitable approximation of the input.

The paper is organized as follows. Section 2 introduces necessary concepts and results from computable analysis and the theory of ODEs. Section 3 presents a theorem stating that the maximal interval $(\alpha, \beta) \subset \mathbb{R}$ of (1), where the solution is defined, is recursively enumerable and the solution is computable there, if the data defining the initial value problem is computable. Section 4 provides a counterexample showing that (α, β) is r.e. but non-computable. Due to the page limit, proofs are either sketchy or omitted.

2 Preliminaries

This section introduces necessary concepts and results from computable analysis and from the theory of ODEs. For more details the reader is referred to [PER89], [Ko91], [Wei00] for computable analysis and [CL55], [Lef65] for ODEs. The idea underlying these definitions is as follows. Consider the number π. This should be a "computable number" from an intuitive point of view, since we can design an algorithm that gives us any number of digits of its decimal expansion. This was the idea of Turing in his seminal paper [Tur36]. However, as he soon recognized, the decimal expansion is not adequate to define computable real functions (for instance, it can be shown [Wei00] that the function $x \mapsto 3x$ is not computable in this framework). Instead, we need a different approach. The classical procedure is based on the following idea. It is known that the set $\mathbb{Q}$ of rational numbers is dense in $\mathbb{R}$. Hence, each real can be approximated by a sequence of rational numbers. Therefore, if one can find an algorithm that, given some precision 2^{-n} as input (i.e. we want the output to have n significant bits), gives a rational q satisfying $|q - x| < 2^{-n}$, one says that $x \in \mathbb{R}$ is computable. We now precise this and other notions.

Definition 1. *1. A sequence $\{r_n\}$ of rational numbers is called a ρ-name of a real number x if there are four functions a, b, c, d from $\mathbb{N}$ to $\mathbb{N}$, where $\mathbb{N}$ denotes the set of natural* *numbers including* 0*, such that for all* $n \in \mathbb{N}$, $r_n = (-1)^{a(n)}\frac{b(n)}{c(n)+1}$ *and*

$$j > d(n) \quad \Rightarrow \quad |r_j - x| \leq \frac{1}{2^n}. \tag{2}$$

2. *A real number* x *is called computable if* a, b, c *and* d *are computable (recursive) functions.*
3. *A sequence* $\{x_k\}_{k\in\mathbb{N}}$ *of real numbers is computable if there are four computable functions* a, b, c, d *from* $\mathbb{N}^2$ *to* $\mathbb{N}$ *such that, for all* $k, n \in \mathbb{N}$,

$$j > d(k,n) \quad \Rightarrow \quad \left|(-1)^{a(j,k)}\frac{b(j,k)}{c(j,k)+1} - x_k\right| \leq \frac{1}{2^n}.$$

Similarly, we can define computable points and sequences over $\mathbb{R}^m$, $m > 1$, by assuming that each component is computable. Next we present a notion of computability for open and closed subsets of $\mathbb{R}^m$, which can be found in [Wei00].

Definition 2. *1. An open set* $E \subseteq \mathbb{R}^m$ *is called recursively enumerable (r.e. for short) open if there are computable sequences* $\{a_n\}$ *and* $\{r_n\}$, $a_n \in E \cap \mathbb{Q}^m$ *and* $r_n \in \mathbb{Q}$ *such that*

$$E = \cup_{n=0}^{\infty} B(a_n, r_n)$$

and for any $n \in \mathbb{N}$*, the closure of* $B(a_n, r_n)$*, denoted as* $\overline{B(a_n, r_n)}$*, is contained in* E*, where* $B(a_n, r_n) = \{x \in \mathbb{R}^m : |x - a_n| < r_n\}$.

2. *A closed subset* $K \subseteq \mathbb{R}^m$ *is called r.e. closed if there exist computable sequences* $\{b_n\}$ *and* $\{s_n\}$, $b_n \in \mathbb{Q}^m$ *and* $s_n \in \mathbb{Q}$*, such that* $\{B(b_n, s_n)\}_{n\in\mathbb{N}}$ *lists all rational balls intersecting* K.
3. *An open set* $E \subseteq \mathbb{R}^m$ *is call computable (or recursive) if* E *is r.e. open and its complement is r.e. closed.*

It is well known that a bounded open interval $(\alpha, \beta) \subset \mathbb{R}$ is computable if and only if α and β are computable real numbers. Throughout the paper we assume that $E \subseteq \mathbb{R}^{m+1}$ is r.e. open, $\{a_n\}$, $\{r_n\}$ and $B(a_n, r_n)$ are defined as above. Let $C^k(E)$ denote the set of all continuously differentiable, up to order k, functions defined on E.

Definition 3. *A function* $f : E \to \mathbb{R}^m$ *is computable if there is a Type-2 Turing machine which translates any input* ρ*-name of* $x \in E$ *to a* ρ*-name of* $f(x)$*. Or equivalently, there is an oracle Turing machine such that for any input* $n \in \mathbb{N}$ *(accuracy) and any* ρ*-name of* $x \in E$ *given as an oracle, the machine will output a rational number* r *satisfying* $|r - x| \leq 2^{-n}$.

We recall that a Type-2 Turing machine is an ordinary Turing machine that allows infinite sequences of symbols from a finite alphabet as input as well as output (see, for example, [Wei00] for more details). It can be proved that if f is computable on E, then there exists a computable modulus function $e : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is locally effective in the sense that $|f(x) - f(y)| \leq 2^{-k}$ whenever $x, y \in \cup_{j=0}^{n}\overline{B(a_j, r_j)}$ and $|x - y| \leq 2^{-e(k,n)}$. In particular, this implies that f must be continuous.

Recall that a function $f : E \to \mathbb{R}^m$ is said to be locally Lipschitz if f satisfies a Lipschitz condition on every compact set $V \subset E$. The following definition gives a computable analysis analog of the Lipschitz condition.

Definition 4. *Let $E = \cup_{n=0}^{\infty} B(a_n, r_n)$ be a r.e. open set. A function $f : E \to \mathbb{R}^m$ is called effectively locally Lipschitz on E if there exists a computable sequence $\{K_n\}$ of positive integers such that*

$$|f(x) - f(y)| \leq K_n |x - y| \text{ whenever } x, y \in \overline{B(a_n, r_n)}.$$

Definition 5. *Let $E = \cup_{n=0}^{\infty} B(a_n, r_n)$ be a r.e. open set. A function $f : E \to \mathbb{R}^m$, with $E \subseteq \mathbb{R}^{j+1}$ is effectively locally Lipschitz on the last j variables if there exists a computable sequence $\{K_n\}$ of positive integers such that if t denotes the first variable of f, then*

$$|f(t, x) - f(t, y)| \leq K_n |y - x| \text{ whenever } (t, x), (t, y) \in \overline{B(a_n, r_n)}.$$

Notice that an effectively locally Lipschitz function $f : E \to \mathbb{R}^m$, where $E \subseteq \mathbb{R}^{m+1}$, is also effectively locally Lipschitz on the m last variables. Also it is clear that if $f \in C^1(E)$ then it is locally Lipschitz on E, so that continuous differentiability is a strictly stronger condition than being locally Lipschitz. This fact extends to computable functions in the following way.

Theorem 6. *Assume that E is r.e. open and that $f : \mathbb{R}^m \to \mathbb{R}^k$ is a computable function in $C^1(E)$ (meaning that both f and its derivative f' are computable). Then f is effectively locally Lipschitz on E.*

Proof. Let K_n be an integer greater or equal to $\max_{x \in \overline{B(a_n, r_n)}} |f'(x)|$. Since f', a_n, r_n are computable, the real number $\max_{x \in \overline{B(a_n, r_n)}} |f'(x)|$ is also computable (notice that, because f' is computable, it has a modulus of continuity that can be used to get the maximum over $\overline{B(a_n, r_n)}$ within any preassigned precision). Moreover, since f' has a locally effective modulus of continuity, subsequently we may assume that the sequence $\{K_n\}$ is a computable sequence of positive integers. Now, for any $x, y \in \overline{B(a_n, r_n)}$, let $u = y - x$. Then $x + su \in \overline{B(a_n, r_n)}$ for $0 \leq s \leq 1$ because $\overline{B(a_n, r_n)}$ is a convex set. Define $F : [0, 1] \to \mathbb{R}^n$ by $F(s) = f(x + su)$. Then by the chain rule,

$$F'(s) = f'(x + su) \cdot u = f'(x + su) \cdot (y - x).$$

Therefore,

$$|f(x) - f(y)| = |F(1) - F(0)| = \left| \int_0^1 F'(s) ds \right| \leq \int_0^1 |f'(x + su) \cdot (y - x)| ds \leq K_n |x - y|.$$

□

Next we turn our attention to some results concerning initial value problems defined with ODEs. Let us consider the following initial value problem

$$\begin{cases} \dot{x} = f(t, x), \\ x(t_0) = x_0, \end{cases} \tag{3}$$

where $(t_0, x_0) \in E \subset \mathbb{R}^{m+1}$ and $f : E \to \mathbb{R}^m$ is a continuous function and satisfies a local Lipschitz condition in the second variable. The following is an immediate consequence of the fundamental existence-uniqueness theory for the initial value problem (3) [CL55], [Lef65].

Theorem 7 (Maximal interval of existence). *Let E be an open subset of $\mathbb{R}^{m+1}$ and assume that $f : E \to \mathbb{R}^m$ is continuous and locally Lipschitz in the second argument. Then for each $(t_0, x_0) \in E$, the problem (3) has a unique solution $x(t)$ defined on a maximal interval (α, β) with the following property: if $\beta < \infty$, either $(t, x(t))$ approaches the boundary of E, or $x(t)$ is unbounded as $t \to \beta^-$ (similar conditions hold for α).*

3 Computability of the maximal interval

Theorem 8. *Let $E \subseteq \mathbb{R}^{m+1}$ be a r.e open set and $f : E \to \mathbb{R}^m$ be a computable function that is also effectively locally Lipschitz on the last m variables. Let (α, β) be the maximal interval of existence of the solution $x(t)$ of the initial-value problem (3) where (t_0, x_0) is a computable point in E. Then (α, β) is a r.e. open interval and x is a computable function on (α, β).*

Proof. (Sketch) We consider the right maximal interval (t_0, β) and prove (t_0, β) is r.e. open and x is computable on it. The same argument applies to the left maximal interval (α, t_0). For simplicity, we assume that E is an open subset of 2-dimensional Euclidean space $\mathbb{R}^2$.

Since $\{a_n\}$ and $\{r_n\}$ are computable sequences and f is a computable function on E, both sequences $\{M_n\}$, $M_n = \max_{z \in \overline{B(a_n, r_n)}} |f(z)|$, and $\{K_n\}$, as defined in Def. 5 are computable. Then, using the classical proof of the existence of the solution of a given ODE (*cf.* [CL55], [Lef65]), one can use the following algorithm to compute the maximal interval

```
1. Set $n = 0$
2. Compute an index $l_n$ such that $x_n \in B(a_{l_n}, r_{l_n})$
3. Compute a time interval $[t_n, t_{n+1}]$ where the solution of $\dot{x} = f(t, x)$, $x(t_n) = x_n$ is defined
4. Set $x_{n+1} = x(t_{n+1})$ and increment $n$
5. Go to step 2
```

Notice that the time interval $[t_n, t_{n+1}]$ referred to in step 3 can be obtained from the proof of the existence of the solution of a given ODE. For instance, one can take $t_{n+1} = t_n + \min\{2^{-K_{l_n}}/M_{l_n}, 2^{-K_{l_n}}\}$ [CL55], [Lef65]. Then it is possible to show, by a contradiction argument, that the maximal interval is given by $(t_0, \beta) = \cup_{n=0}^{\infty}(t_0, t_n)$, i.e. that $t_n \to \beta$ as $n \to \infty$. The idea is the following. If t_n does not converge to β, it must converge to some $\gamma < \beta$. Then, for some index $j \in \mathbb{N}$, one has $(\gamma, x(\gamma)) \in B(a_j, r_j)$. Since t_n converges increasingly to γ and it is known classically that $x : [t_0, \beta) \to E$ is continuous, one can find a sufficiently large n_0 such that $(t_{n_0}, x(t_{n_0})) \in B(a_j, r_j)$ and $t_{n_0} + \min\{2^{-K_j}/M_j, 2^{-K_j}\} > \gamma$. But, by construction, $t_{n_0+1} = t_{n_0} + \min\{2^{-K_j}/M_j, 2^{-K_j}\}$. Thus $t_{n_0+1} > \gamma$. We have a contradiction.

The solution x is computable because, given $t \in (t_0, \beta)$, we can: (i) get an index $n \in \mathbb{N}$ such that $t \leq t_n$ (ii) compute numerically the solution over $[t_0, t_n]$ to get the value of $x(t)$. □

4 Non-recursiveness of the maximal interval

In this section, we present some undecidability results concerning ODEs. In particular, for the initial-value problem (3), we show that the maximal interval can be non-computable. Although our result is for the case where f is continuous (more precisely, f is piecewise linear), nevertheless, the construction can be "smoothed" so that f becomes C^∞. Moreover, an explicit expression can be written for such an f. The construction depends on the following lemma, which can also be used to prove other results concerning undecidability.

Lemma 9. *Let $a : \mathbb{N} \to \mathbb{N}$ be a computable function. Then there exists a computable and effectively locally Lipschitz function $f : \mathbb{R} \to \mathbb{R}$ such that the unique solution of the problem*

$$\dot{x} = f(x), \ x(0) = 0 \tag{4}$$

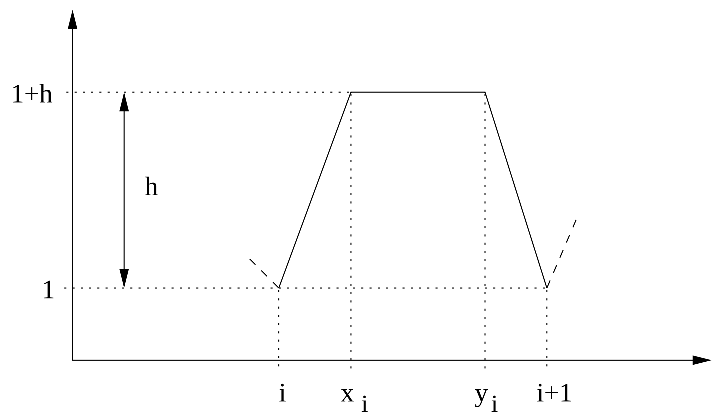


Fig. 1. Sketch of the function f on the interval $[i, i+1]$, for $i \in \mathbb{N}$.

is defined on a maximal interval $(-\alpha, \alpha)$ *with*

$$\alpha = \sum_{i=0}^{\infty} \frac{1}{2^{a(i)}}.$$

Proof. (Sketch) The idea is as follows: f is constructed piecewisely on intervals of the form $[i, i+1]$, $i \in \mathbb{N}$ (for negative values, we take $f(x) = f(|x|)$) in such a way that the solution of the initial-value problem

$$\dot{x} = f(x), \; x(0) = i \tag{5}$$

satisfies $x(2^{-a(i)}) = i+1$, which implies that the solution of the problem $\dot{x} = f(x)$ and $x(0) = 0$ will satisfy $x(2^{-a(0)}) = 1$, $x(2^{-a(0)} + 2^{-a(1)}) = 2$, ..., or more generally

$$x\left(\sum_{i=0}^{n} 2^{-a(i)}\right) = n+1, \quad \text{for all } n \in \mathbb{N}.$$

Notice that f does not depend on t and therefore the solution is invariant under time translations. If we take $\alpha = \sum_{i=0}^{\infty} 2^{-a(i)}$, then $x(t) \to \infty$ as $t \to \alpha^-$. For $t < 0$, we require that $x(-2^{-a(i)}) = -(i+1)$, then $x(t) \to -\infty$ as $t \to -\alpha^+$. Therefore the maximal interval must be $(-\alpha, \alpha)$.

The function f is defined on each interval $[i, i+1]$ as suggested by Fig. 1. Since f must be continuous, we need to glue the values of f at the endpoints of these intervals. This is achieved by assuming that $f(i) = 1$ for $i \in \mathbb{N}$. It can be shown that if we define the function f on the interval $[i, i+1]$ as follows:

$$f(x) = \begin{cases} 1 + (x-i)2^{a(i)}/(x_i - i) & \text{if } x \in [i, x_i) \\ 1 + 2^{a(i)} & \text{if } x \in [x_i, y_i), \\ 1 + 2^{a(i)} - (x - y_i)2^{a(i)}/(i + 1 - y_i) & \text{if } x \in [y_i, i+1), \end{cases}$$

where

$$x_i = i + \frac{1 - \Delta_i}{2}, \qquad y_i = i + \frac{1 + \Delta_i}{2},$$

and

$$0 < \Delta_i = \frac{2^{-a(i)} - 2^{-a(i)} \ln(2^{a(i)} + 1)}{(1 + 2^{a(i)})^{-1} - 2^{-a(i)} \ln(2^{a(i)} + 1)} < 1,$$

the solution of (5) satisfies $x(2^{-a(i)}) = i + 1$ as requested. Moreover, this f is easily seen to be computable and effectively locally Lipschitz. □

Theorem 10. *There exists an effectively locally Lipschitz computable function $f : \mathbb{R} \to \mathbb{R}$ such that the unique solution of the problem*

$$\dot{x} = f(x),\ x(0) = 0$$

is defined on a non-computable maximal interval.

Proof. In [PER89, Sec. 0.2], it is shown that if $a : \mathbb{N} \to \mathbb{N}$ is a one to one recursive function generating a recursively enumerable nonrecursive set A, then $\alpha = \sum_{i=0}^{\infty} 2^{-a(i)}$ is a non computable real number. Consequently, the open interval $(-\alpha, \alpha)$ is non-computable. The theorem now follows immediately from the previous lemma. □

The function f in Theorem 10 can be constructed so that f is of class C^∞ and all its derivatives are computable functions, and hence f is also effectively locally Lipschitz. This condition matches the assumption set down in Theorem 8. Thus, Theorem 8 gives rise to the best possible result concerning computability of a maximal interval.

5 Conclusion

In this paper we studied some computational issues regarding Initial Value Problems defined with ODEs. In particular we showed that IVPs (1), where f and (t_0, x_0) are recursive can have a nonrecursive maximal interval, thus suggesting some fundamental limitations on the design of numerical methods for solving ODEs. Note that this result is valid for the case when f is of class C^∞, but we didn't cover the case where f is analytic. It would be interesting to know what happens in this case.

Acknowledgments. DG wishes to thank Manuel Campagnolo for helpful discussions. DG was partially supported by the *Fundação Calouste Gulbenkian* through the *Programa Gulbenkian de Estímulo à Investigação*, and by the *Fundação para a Ciência e a Tecnologia* and EU FEDER POCTI/POCI via CLC, grant SFRH/BD/17436/2004, and the project ConTComp POCTI/MAT/45978 /2002. NZ was partially supported by University Cincinnati's Taft Summer Research Fellowship. JB was partially supported by *Fundação para a Ciência e a Tecnologia* and EU FEDER POCTI/POCI via CAMGSD.

References

[Abe70] O. Aberth. Computable analysis and differential equations. In A. Kino, J. Myhill, and R. E. Vesley, editors, *Intuitionism and Proof Theory*, Studies in Logic and the Foundations of Mathematics, pages 47–52. North-Holland, 1970.

[Abe71] O. Aberth. The failure in computable analysis of a classical existence theorem for differential equations. *Proc. Amer. Math. Soc.*, 30:151–156, 1971.

[BB85] E. Bishop and D. S. Bridges. *Constructive Analysis.* Springer, 1985.

[CL55] E. A. Coddington and N. Levinson. *Theory of Ordinary Differential Equations.* Mc-Graw-Hill, 1955.

[Ko91] K.-I Ko. *Computational Complexity of Real Functions.* Birkhäuser, 1991.

[Lef65] S. Lefshetz. *Differential equations: Geometric theory.* Interscience, 2nd edition, 1965.

[PER79] M. B. Pour-El and J. I. Richards. A computable ordinary differential equation which possesses no computable solution. *Ann. Math. Logic*, 17:61–90, 1979.

[PER81] M. B. Pour-El and J. I. Richards. The wave equation with computable initial data such that its unique solution is not computable. *Adv. Math.*, 39:215–239, 1981.

[PER89] M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics.* Springer, 1989.

[Ruo96] K. Ruohonen. An effective cauchy-peano existence theorem for unique solutions. *Internat. J. Found. Comput. Sci.*, 7(2):151–160, 1996.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[Wei00] K. Weihrauch. *Computable Analysis: an Introduction.* Springer, 2000.

[WZ02] K. Weihrauch and N. Zhong. Is wave propagation computable or can wave computers beat the Turing machine? *Proc. London Math. Soc.*, 85(3):312–332, 2002.

Interval Analysis Without Intervals (extended abstract)

Paul Taylor

`http://www.cs.man.ac.uk/~pt/ASD`

Manchester University

Abstract. We argue that Dedekind completeness and the Heine–Borel property should be seen as part of the "algebraic" structure of the real line, along with the usual arithmetic operations and relations. Compactness is expressed as a universal quantifier in a logic of computationally observable properties. Set theory plays no part whatsoever, so cuts are defined in an original way based on this syntactic calculus: they enjoy proof-theoretic introduction and elimination rules similar to those for lambda abstraction, where the arithmetic order plays the role of application. Along with recursion, this completely axiomatises computably continuous functions on the real line.

We begin to exploit the syntactic nature of this calculus (of "single" points) by translating it formally into Interval Analysis, interpreting the arithmetic operations a la Moore, and compactness as optimisation under constraints. Notice that interval computation is the conclusion and not the starting point.

This account of the real line is part of a more general recursive axiomatisation of general topology called Abstract Stone Duality.

Reliable computation gives the authority of a theorem to the results of numerical computations, by setting interval bounds on each step of the calculation. This may be done either within the fixed precision of machine arithmetic, or by allowing this to extend to as many digits as may be required. In either form, this discipline usually fits into the bigger picture of continuous-valued computation by adapting techniques from *numerical* analysis. It often does this by giving them "double vision", *i.e.* computing upper and lower bounds where standard methods would just take whatever rounded value the machine arithmetic happens to provide.

The phrase "without intervals" in the title of this paper signifies that it is part of a research programme (called "Abstract Stone Duality") whose long term motivation is to make a direct link from *theoretical* analysis and topology to computation. Motivated by the view that the various disciplines of exact real computation are overly concerned with minor details of representations, we *begin* from a calculus that naturally and directly axiomatises the computable real line, and work *towards* computational implementation. Specifically, we shall translate a theoretical calculus whose basic entities are *single, exact, real* numbers into a computational one that manipulates intervals with machine representable endpoints in a Prolog-like fashion.

The theoretical benefit of (the availability of) this translation is that it eliminates the double vision of interval analysis, in which interval analogues of single-valued concepts (arithmetic, trigonometry, differentiation, Banach spaces ...) have to be devised *ad hoc*, one at a time, in a fashion that gets increasingly estranged from any theoretical roots. Real analysis can be developed within ASD in a single-valued style that is very similar to the traditional one. For us, intervals are the outcome and not the starting point of the investigation.

Taking a principled theoretical view also helps to restructure and generalise a programming task. When computing numbers to hundreds of decimal places, we come to the point where we discover that thousands were needed instead, and in fact such *back-tracking* may be needed for several other reasons, such as that we need more terms of a power series, or more sample points of an integral, as well as more digits of particular numbers. A unified structure is therefore needed to manage the flow of control. At a higher level, since ASD has already provided an account of locally compact spaces in general, it can show how to design a system of "cells" that generalise intervals on the real line in a way that is appropriate to $\mathbb{R}^{\mathbf{n}}$, the complex (Riemann) sphere or other spaces, instead of just using cubes.

The real line as we understand it in analysis must be "complete". Other approaches both to exact computation and in constructive logics typically formulate this property as convergence of Cauchy sequences. We advocate *Dedekind* completeness instead. We demonstrate how this simplifies the formulation of derivatives, integrals and power series. We also show how Dedekind cuts generalise to intervals, in a way that arises naturally for these basic ideas of analysis.

The many schools of computable or constructive analysis accept without question the received notion of set with structure. (In doing so they blatantly disregard the facts of history, that Cantor invented set theory in 1870, after most of the real analysis that engineers and scientists now use was already well established.) These schools rein in the wild behaviour of set-theoretic functions using the double bridle of topology and recursion theory, adding encodings of explicit numerical representations to the epsilons and deltas of metrical analysis. Fundamental conceptual results such as the Heine–Borel theorem can only be saved by set-theoretic tricks such as Turing tapes with infinitely many non-trivial symbols.

It doesn't have to be like that.

When studying computable continuous functions, we should never introduce uncomputable or discontinuous ones, only to exclude them later. By the analogy between topology and computation, we also concentrate on open subspaces. So we admit $+$, $-$, $\times$, $\div$, $<$, $>$, $\neq$, $\wedge$ and $\vee$, but not $\leqslant$, $\geqslant$, $=$, $\neg$ or $\Rightarrow$.

Universal quantification ($\forall$) captures the Heine–Borel theorem, being allowed over *compact* spaces (closed bounded intervals). It is very important to understand, however, that we must reason with $\forall$ according to its proof-theoretic rules, and not try to interpret it as meaning "for every computably definable real number" — since the latter are enumerable, we may cover them with diminishing intervals of which no finite subset would suffice. It is a little annoying for analysis that we are not allowed to write $\forall\epsilon > 0$ or $\forall n$ (since $(0, \infty)$ and $\mathbb{N}$ are not compact), but these things can be said using free variables instead. We stress that this restriction is intended, and that our calculus is not some impoverished version of first order logic that aspires one day to the full strength.

Dedekind completeness is usually formulated using set theory, even in Bishop-style constructive analysis. For us, the two halves of the cut are given instead by predicates in the language that we have just described, and completeness is stated in the form of introduction and elimination rules in the style of proof theory, where the arithmetic order plays the role of λ-application. The manner in which we state Dedekind completeness would be enough on its own to make this work original.

If interval analysis really did consist only in stating upper and lower bounds of the successive stages of a computation, it would be of very little use, since these intervals rapidly become so large that the precise value is entirely lost. What makes interval methods so powerful is that crude arithmetical calculations provide rather strong *logical* information about the behaviour of a function over a finite range.

For example, the *Interval Newton* algorithm evaluates a function at a point in the middle of an interval and its derivative on the whole interval using Moore's arithmetic. The result of this is to constrain the values of the function at *all* of the points in the interval. In particular, if the function has a positive value at the chosen point, it is excluded from a certain triangle under that point, and there can be no zero in a large segment of the interval. Like the original Newton algorithm, this doubles the number of bits of precision at each iteration, in the case where the function is twice differentiable and has just one zero in the given interval. However, whereas its classical counterpart behaves chaotically when these conditions fail, the Interval Newton algorithm merely puts infinite (*i.e.* no) bounds on the derivative, and so degrades gracefully into interval halving, in order to separate the zeroes before finding them precisely.

This example demonstrates that *the Moore interpretation is an approximation to* $\forall$, — so we write ${\forall\!\!\!\mid}$ for it (a cross between $\forall$ and M).

That is, for any predicate ϕx with a real variable x in our language, we write ${\forall\!\!\!\mid} x \in \mathsf{x}.\, \phi x$ for the *syntactic translation* that replaces x by an interval variable x and the real arithmetic operations $+-\times\div$ and relations $<>\neq$ by their Moore counterparts $\oplus\ominus\otimes\oslash\olessthan\ogreaterthan\pitchfork$. The logical connectives $\top\bot\wedge\vee$ and $\exists n : \mathbb{N}$ remain the same, whilst $\exists x : \mathbb{R}$ becomes $\exists \mathsf{x}$. The universal quantifier $\forall x \in [d, u]$, on the other hand, must be replaced by a program that is based on the idea of the Heine–Borel theorem.

This translation is given a logical meaning by the equivalence

$$\phi x \iff \exists \mathsf{x}.\, x \in \mathsf{x} \wedge \forall y \in \mathsf{x}.\, \phi y \iff \exists \mathsf{x}.\, x \in \mathsf{x} \wedge {\forall\!\!\!\mid} y \in \mathsf{x}.\, \phi y,$$

which is proved by structural induction on the formula ϕ.

In its $\forall$-form, this states *local compactness* of $\mathbb{R}$. The predicate ϕx says in logical notation that the point $x \in \mathbb{R}$ belongs to the open subspace $U \subset \mathbb{R}$ that is defined by ϕ. Then there is a compact interval $[\mathsf{x}]$ that is contained in U (we write $\forall y \in \mathsf{x}.\, \phi y$), and which contains the point x in its interior, (x).

The Moore translation ${\forall\!\!\!\mid} y \in \mathsf{x}.\, \phi y$ is obtained by evaluating the arithmetic operations that occur in the formula ϕ according to interval arithmetic, with the interval y in place of the singleton real argument y. This over-estimates the image, a fact that we write in logical notation as

$${\forall\!\!\!\mid} y \in \mathsf{x}.\, \phi y \;\Rightarrow\; \forall y \in \mathsf{x}.\, \phi y.$$

This statement, in the special case where $\phi x \equiv |fx - a| < \epsilon$ for a function $f : \mathbb{R} \to \mathbb{R}$, is sometimes called the *Fundamental Theorem of Interval Analysis*, but, as I have said, this subject would be useless if it amounted to no more than this. The title of "fundamental theorem" should therefore be conferred on the stronger result above. Although the interval x in the ${\forall\!\!\!\mid}$-statement may have to be narrower than that in the $\forall$-statement, the former can be used with the same logical strength, despite being based on a crude piece of arithmetic.

Before giving examples of the use of the Fundamental Theorem, let alone saying how to compute with it, we need to understand a little more about the topological meaning of the cellular notation. We have already seen that the cell x is sometimes treated as a compact subspace $[\mathsf{x}]$ and sometimes as an open one (x). Indeed, this ambiguity is also to be found in the literature on the applications of interval analysis and at first sight seems mathematically rather casual. In fact, $[\mathsf{x}]$ and (x) both play important, *dual* roles:

- the *compact* interval $[\mathsf{x}]$ states the precision of an *input* value, as well as the range of a universal quantifier, so
$$\forall x \in \mathsf{x} \quad \text{means} \quad \forall x \in [\mathsf{x}],$$
- whilst the *open* interval (x) is a predicate (test), so
$$x \in \mathsf{x} \quad \text{means} \quad x \in (\mathsf{x}),$$
and receives *output* values, and
- existentially quantified variables $(\exists \mathsf{x})$ are used for *communication* ($\exists x \in (\mathsf{x})$ and $\exists x \in [\mathsf{x}]$ are equivalent).

These roles are linked by the "way inside" relation
$$\mathsf{x} \Subset \mathsf{y}, \quad \text{which means} \quad [\mathsf{x}] \subset (\mathsf{y}),$$
whose importance we learn from the theory of continuous lattices that lies behind local compactness. It is by considering the geometry of compact and open pairs of cells (and in particular how a finite intersection of compact cells is covered by a finite union of open ones) that we see how to generalise interval analysis effectively from $\mathbb{R}$ to other locally compact spaces.

Now consider a function $f : \mathbb{R} \to \mathbb{R}$. Evaluating it to precision ϵ on an argument a means that we find an interval y with
$$f(a) \in \mathsf{y} \quad \text{and} \quad \|\mathsf{y}\| < \epsilon,$$
but the fundamental theorem says that
$$f(a) \in \mathsf{y} \iff \exists \mathsf{x}.\, a \in \mathsf{x} \wedge \forall\!\!\!\forall x \in \mathsf{x}.\, f(x) \in \mathsf{y}$$
where $\|\mathsf{x}\| < \delta$. Introducing a notation similar to λ-abstraction for Dedekind cuts, we recover the function as
$$f(a) \;=\; \mathsf{cut}\,\mathsf{y}.\, \exists \mathsf{x}.\, a \in \mathsf{x} \wedge \forall\!\!\!\forall x \in \mathsf{x}.\, f(x) \in \mathsf{y}.$$
Let us substitute this formula into a similar one for $g(b)$:
$$\begin{aligned} g\big(f(a)\big) &= \mathsf{cut}\,\mathsf{z}.\, \exists \mathsf{y}.\, \mathsf{cut}\,\mathsf{y}'.\, \big(\exists \mathsf{x}.\, a \in \mathsf{x} \wedge \forall\!\!\!\forall x \in \mathsf{x}.\, fx \in \mathsf{y}'\big) \in \mathsf{y} \wedge \forall\!\!\!\forall y \in \mathsf{y}.\, gy \in \mathsf{z} \\ &= \mathsf{cut}\,\mathsf{z}.\, \big(\exists \mathsf{y}.\, \exists \mathsf{x}.\, a \in \mathsf{x} \wedge \forall\!\!\!\forall x \in \mathsf{x}.\, fx \in \mathsf{y}\big) \wedge \forall\!\!\!\forall y \in \mathsf{y}.\, gy \in \mathsf{z}, \end{aligned}$$
using the β-rule for cuts: notice that this swallows part of the context, namely "$\in \mathsf{y}$", treating it in the same way that the β-rule of the λ-calculus does the argument of a function.

Even though logical conjunction $(\wedge)$ is commutative, the input–output roles as described above set up a flow of data from x to y to z. When merely evaluating expressions, we only find (output) variables on the right hand side of $\Subset$.

An equation $f(x) = 0$ is not as it stands a valid predicate in our language, because this cannot test equality of real numbers. We say that x is a *stable zero* if, arbitrarily close to it, there are $x^- < x < x^+$ with $f(x^-) < 0 < f(x^+)$ or $f(x^-) > 0 > f(x^+)$. Under the translation, there are intervals $\mathsf{x}^- \mathrel{\lessdot} \mathsf{x}^+$ with
$$\forall\!\!\!\forall x^- \in \mathsf{x}^-.\, \forall\!\!\!\forall x^+ \in \mathsf{x}^+.\, f(x^-) < 0 < f(x^+) \;\vee\; f(x^-) > 0 > f(x^+).$$

This implies (but is not implied by) the existence of an interval x that contains x^-, x and x^+ and for which

$$0 \in f(\mathsf{x}) \equiv |\!\forall\!| x' \in \mathsf{x}.\, f(x'),$$

where we use both $f(\mathsf{x})$ and $|\!\forall\!|$ for the Moore interpretation of arithmetic as well as logical expressions.

So when solving equations, we find interval expressions on the right hand side of $\Subset$ (which is the translation of $\in$), but it is still the variable (contained in the expression) on that side that is the required output.

Even though the fundamental theorem has eliminated individual real numbers in favour of machine-representable intervals, it still leaves a logical expression and not a program. We *execute* such expressions by trying to find *proofs* of them, as in Prolog. The rules for doing this in the fragment of the language consisting of $\oplus\ominus\otimes\oslash\olessthan\ogreaterthan\pitchfork\wedge$ and $\exists\mathsf{x}$ (but not $\exists n$, $\vee$ or $\forall x \in \mathsf{x}$) were devised by John Cleary and incorporated into some implementations of Prolog.

The procedure is iterative, initially assigning the interval $[-\infty, +\infty]$ to the existentially bound variables. Inside these quantifiers, the expression is a conjunction of $\olessthan$, $\ogreaterthan$, $\pitchfork$ and $\Subset$ relations between (interval) arithmetic expressions, along with specifications of the required output precision in the form $\|\mathsf{y}\| < \epsilon$. If this conjunction evaluates to $\top$, we are done. Otherwise, if we wanted $\mathsf{x} \olessthan \mathsf{y}$ (*i.e.* x is to lie wholly to the left of y) but in fact $\mathsf{x} \ogreaterthan \mathsf{y}$ then we have *logical failure*. On the other hand, if x and y overlap then we trim the right-hand end of x and left-hand end of y to make the situation *consistent* in the sense that there exist $x \in \mathsf{x}$, $y \in \mathsf{y}$ with $x < y$.

If this doesn't achieve definite logical success or failure (in particular if some conjunct $\mathsf{x} \pitchfork \mathsf{y}$, $\mathsf{x} \Subset \mathsf{y}$ or $\|\mathsf{y}\| < \epsilon$ is unsatisfied), then we have to split one or more of the intervals, and apply the same techniques disjunctively to the parts. However, in the case of $\mathsf{x} \Subset \mathsf{y}$, the Interval Newton algorithm usually reduces the parts to considerably less than half of their original size.

The separate consideration of the parts of intervals, disjunction, $\exists n$ and recursion are handled using logical back-tracking as in Prolog: if one disjunct leads to logical failure, we try the next.

The interpretation of $\forall$ is, as we have said, based on the Heine–Borel theorem, and naturally goes with that of disjunction. First we consider each of the disjuncts in turn, trying to prove it using $|\!\forall\!|$ (for the whole interval) in place of $\forall$. If this fails, we try to do it for the left half of the interval, then the leftmost quarter, and so on, dealing with the remainder of the interval in the same way. Of course, we may need to use one disjunct for the first subinterval, another for the second, and so on. The Heine–Borel theorem says that, if the original $\forall$ formula was provable at all, then some such finite subdivision is also provable, using $|\!\forall\!|$ on the parts in place of $\forall$.

In conclusion, we have sketched how a combination of the techniques of topology (locally compact spaces), real analysis, proof theory, lambda calculus, constraint logic programming, interval analysis and high precision real arithmetic can, in principle, lead to the computational interpretation of a language for real analysis that is very similar to the way in which it is formulated in the pure mathematical textbooks.

Conditional Speculative Decimal Addition[⋆]

Álvaro Vázquez and Elisardo Antelo

Department of Electronic and Computer Engineering
University of Santiago de Compostela, Spain
{alvaro,elisardo}@dec.usc.es

Abstract. Financial and e–commerce servers require hardware decimal arithmetic to satisfy both precision and performance demands. In fact, the revision of the IEEE–754 standard for floating–point arithmetic will incorporate a specification for decimal arithmetic. In this paper we present a hardware module to perform decimal addition/subtraction. Moreover, it also performs these operations for binary representation. We describe the algorithm and its architecture and perform a rough comparison with patented and commercial implementations currently in use in industrial designs [1–4, 6]. The rough evaluations performed show that the proposed algorithm potentially leads to adders with interesting area–delay figures.

1 Introduction

Hardware support for decimal arithmetic is of interest for server processors dedicated to financial and e–commerce applications. Some manufactures already implement hardware decimal units in their high–end products, such as IBM in their S/390 mainframe microprocessors (G4, G5 and G6 [4], z900 and z990 [2, 3]). Furthermore, there have been some efforts in defining a standard for decimal floating–point arithmetic [5], and the draft revision of the IEEE–754 and IEEE–854 floating–point standards already incorporate specifications for decimal arithmetic [7]. So future processors are expected to gradually incorporate hardware support for decimal floating–point arithmetic.

In this paper we deal with fixed–point decimal addition/subtraction. This hardware module can be used in both integer or floating–point units. Moreover, we are interested in a unit that allows both binary and decimal addition at high performance.

The structure of the paper is as follows. Section 2 outlines representative methods to compute fixed–point decimal addition/subtraction in hardware. In Section 3 we present a new method to compute decimal addition/subtraction. Evaluation results are shown in Section 4 using a model based on logical effort [11]. We also provide a comparison among the different methods for some representative adder architectures. Finally, the conclusions are summarized in Section 5.

2 Previous Work on Decimal Addition

In this work we use decimal digits coded in standard BCD–8421 code as

$$A_i = \sum_{j=0}^{3} a_i[j] \cdot 2^j \tag{1}$$

where $A_i \in [0, 9]$ is the i^{th} decimal digit and $a_i[j] \in \{0, 1\}$ is the j^{th} bit of the BCD digit i. Decimal subtraction corresponds to the addition of the 10's complement of the subtrahend

[⋆] This work was supported in part by Xunta de Galicia under grant PGIDT03TIC10502PR.

```
[Algorithm: Basic Decimal Addition/Subtraction(S=A±B)]

Inputs: $A = \sum_{i=0}^{d-1} A_i \cdot 10^i, B = \sum_{i=0}^{d-1} B_i \cdot 10^i$
        $C_0 = \begin{cases} 1 & \texttt{If(op == sub)} \\ C_{in} & \texttt{Else} \end{cases}$

For (i=0;i<d;i++){
        $B_i^* = \begin{cases} \overline{B_i + 6} & \texttt{If(op == sub)} \\ B_i & \texttt{Else} \end{cases}$

        $S_i^* = \mathrm{mod}_{16}(A_i + B_i^* + C_i)$

        $C_{i+1} = \lfloor (A_i + B_i^* + C_i)/10 \rfloor = \lfloor (S_i^* + 6)/16 \rfloor$

        $S_i = \mathrm{mod}_{10}(A_i + B_i^* + C_i) = \begin{cases} \mathrm{mod}_{16}(S_i^* + 6) & \texttt{If } C_{i+1} == 1 \\ S_i^* & \texttt{Else} \end{cases}$
}
```

Fig. 1. Basic Algorithm.

operand. The 10's complement is obtained by bit complementing each decimal digit plus 6 (modulo 16) and adding a one in form of carry input to the addition [2].

The basic carry propagate algorithm for decimal addition/subtraction is described in Figure 1. The conditional addition of 6 in each digit position allows to use binary 4–bit carry–propagate adders for each decimal digit. There are several ways to implement this algorithm in hardware. However its main limitation is, obviously, the carry chain. To improve the delay in the computation of the carries, there have been proposed several refinements to the basic algorithm. The best known and more representative high–performance techniques proposed are grouped in two types of methods: *Direct Decimal Addition* [4, 10] and *Decimal Speculative Addition* [1–3, 6, 12].

Direct Decimal Addition is based on the basic algorithm but uses the carry recurrence $C_{i+1} = G_i + \overline{K_i} C_i$. This allows the use of conventional parallel carry evaluation techniques such as parallel prefix. G_i and K_i are the decimal carry–generate and carry–kill functions, which can be expressed in terms of the binary carry–generate $(g_i[j] = a_i[j] + b_i[j])$ and carry–kill signals $(k_i[j] = \overline{a_i[j] + b_i[j]})$ as $G_i = G_i^* + \overline{K_i^*} \cdot g_i[0]$, and $\overline{K_i} = \overline{K_i^*} \cdot \overline{k_i[0]}$, where

$$G_i^* = g_i[3] + g_i[2] \cdot g_i[1] + \overline{k_i[3]} \cdot (\overline{k_i[2]} + \overline{k_i[1]})$$

and

$$\overline{K_i^*} = \overline{k_i[3]} + g_i[2] + \overline{k_i[2]} \cdot g_i[1]$$

Since there is a decimal carry per each 4–bit BCD digit, an appropriate scheme for fast parallel carry evaluation is a quaternary–tree configuration [9]. Decimal adders using direct decimal addition are implemented in the functional units of the G4, G5 and G6 S/390 microprocessors [4]. In Section 4 we evaluate an implementation of this algorithm.

Decimal Speculative Addition unconditionally add 6 to each decimal digit position and then correct the sum digit (subtracting 6) if the carry out from that decimal position is zero. The algorithm for speculative addition is shown in Figure 2. Since the decimal carries are generated when $S_i^* > 15$, they have the same value as the corresponding binary carries in the same position. Therefore, for the evaluation of C_{i+1}(decimal carries) any binary carry

```
[Algorithm: Speculative Decimal Addition/Subtraction (S=A±B)]

Inputs: $A = \sum_{i=0}^{d-1} A_i \cdot 10^i, B = \sum_{i=0}^{d-1} B_i \cdot 10^i$
    $C_0 = \begin{cases} 1 & \texttt{If(op == sub)} \\ C_{in} & \texttt{Else} \end{cases}$

For (i=0;i<d;i++){
    $B_i^* = \begin{cases} \overline{B_i + 6} & \texttt{If(op == sub)} \\ B_i & \texttt{Else} \end{cases}$

    $S_i^* = \mathrm{mod}_{16}(A_i + B_i^* + 6 + C_i)$

    $C_{i+1} = \lfloor (A_i + B_i^* + C_i)/10 \rfloor = \lfloor S_i^*/16 \rfloor$

    $S_i = \mathrm{mod}_{10}(A_i + B_i^* + C_i) = \begin{cases} \mathrm{mod}_{16}(S_i^* - 6) & \texttt{If } C_{i+1} == 0 \\ S_i^* & \texttt{Else} \end{cases}$
}
```

Fig. 2. Speculative Algorithm.

evaluation architecture can be used, for instance, a quaternary–tree configuration. For the evaluation of S_i^* there are two possibilities:

– Using a full binary parallel prefix carry network to obtain all the binary carries. Since decimal and binary carries are equal in this algorithm, the speculative sum digits S_i^* can be obtained directly from the XOR operation of input operands bits and binary carries. A post–correction scheme is necessary after carry evaluation to correct the speculative sum digits S_i^* when they are wrong.
– Using a binary quaternary–tree for carry evaluation, the sum digits are pre–evaluated in a parallel pre–sum stage of 4–bit carry–select adders. The decimal carries computed in the quaternary–tree select the right pre–sum. The correction of wrong speculative sum digits can be done in the pre–sum stage for each of the two possible values of the input decimal carry, in parallel with carry computation.

A decimal speculative fixed–point adder using this carry–select scheme was implemented in the IBM z900 and z990 microprocessors [2, 3] (described in more detail in [1, 6]). In Section 4 we evaluate an implementation of this algorithm. A variation [12] computes $A_i + 3$ and $B_i^* = B_i + 3$ for decimal addition or $B_i^* = \overline{B_i + 3} = (9 - B_i) + 3$ for decimal subtraction. Nevertheless, the resultant implementations are similar.

3 Proposed Method: Conditional Speculative Decimal Addition

We propose a method based on speculative addition which uses a conditional speculation to avoid a post–correction after carry evaluation. This leads to a lower dependency on the performance of the selected carry–tree topology, giving the designer more flexibility to choose the adder architecture and area/latency trade–offs. Moreover, the proposed method provides an efficient implementation of a binary/decimal combined unit using any existing binary parallel prefix adder and a few additional hardware.

We use the following notation. For a decimal digit A_i, we call A_i^U the 3 left–most significant bits of the BCD digit. We also define $c_i[1]$ as the binary carry between the lower (least

[Algorithm: Conditional Speculative Decimal Addition/Subtraction (S=A±B)]

```
Inputs: $A = \sum_{i=0}^{d-1} A_i \cdot 10^i$,$B = \sum_{i=0}^{d-1} B_i \cdot 10^i$
        $C_0 = \begin{cases} 1 & \texttt{If(op == sub)} \\ C_{in} & \texttt{Else} \end{cases}$
For (i=0;i<d;i++){
        $B_i^* = \begin{cases} \overline{B_i + 6} & \texttt{If(op == sub)} \\ B_i & \texttt{Else} \end{cases}$

        $s_i[0] = \mathrm{mod}_2(a_i[0] + b_i^*[0] + C_i)$

        $c_i[1] = \lfloor (a_i[0] + b_i^*[0] + C_i)/2 \rfloor$

        $(S_i^*)^U = \begin{cases} \mathrm{mod}_{16}(A_i^U + (B_i^*)^U + 6 + c_i[1] \cdot 2) & \texttt{If } A_i^U + (B_i^*)^U \geq 8 \\ \mathrm{mod}_{16}(A_i^U + (B_i^*)^U + c_i[1] \cdot 2) & \texttt{Else} \end{cases}$

        $S_i = \mathrm{mod}_{10}(A_i + B_i^* + C_i) = \begin{cases} ((S_i^*)^U - 6, s_i[0]) & \texttt{If } (S_i^*)^U == 14 \quad \left(A_i^U + (B_i^*)^U == 8 \text{ and } c_i[1] == 0\right) \\ ((S_i^*)^U, s_i[0]) & \texttt{Else} \end{cases}$

        $C_{i+1} = \lfloor (A_i + B_i^* + C_i)/10 \rfloor = \lfloor ((S_i^*)^U, s_i[0])/16 \rfloor$

}
```

Fig. 3. Proposed Conditional Speculative Algorithm.

significant bit) and upper (the 3 left–most significant bits) part for the evaluation of the sum digit.

Since the condition for the generation of a decimal carry is $S_i^* = A_i + B_i^* + C_i \geq 10$, and we have that $S_i^* - 1 \leq (S_i^*)^U \leq S_i^*$ with $(S_i^*)^U$ even, that condition can be stated as

$$(S_i^*)^U = A_i^U + (B_i^*)^U + c_i[1] \cdot 2 \geq 10 \tag{2}$$

That is,

$$A_i^U + (B_i^*)^U \geq 10 - c_i[1] \cdot 2 \quad (c_i[1] \in \{0, 1\}) \tag{3}$$

resulting in

$$A_i^U + (B_i^*)^U \geq 8 \tag{4}$$

which is a necessary (but not sufficient) condition for the generation of a decimal carry at position i. We use expression (4) for speculation. That is, we add 6 to the digit position if (4) is true. The speculation fails for $A_i^U + (B_i^*)^U = 8$ and $c_i[1] = 0$, so the resulting digit must be corrected in this case (subtraction of 6).

Based on this speculative scheme, we show in Figure 3 the proposed conditional speculative algorithm for decimal addition/subtraction.

The decimal carry evaluation is always correct because the addition of 6 does not produce a decimal carry–out when $(S_i^U)^* = 14$ (that is, $A_i^U + (B_i^*)^U = 8$ and $c_i[1] = 0$), the value for which speculation fails. Since decimal and binary carries have the same value as in the speculative method, conditional speculative addition can be implemented using a full binary parallel prefix adder or a quaternary–tree adder.

To have a simple implementation, we detect the condition $A_i^U + (B_i^*)^U \geq 8$ separately for addition and subtraction. For decimal addition, since $B^* = B$, the condition to be detected is

$A_i^U + B_i^U \geq 8$. We define a control signal r_i which is true when this condition is verified. The value of r_i is obtained in terms of the binary carry–kill and carry–generate functions (using the bits of A_i and B_i) as follows:

$$r_i = \overline{k}_i[3] + g_i[2] + \overline{k}_i[1] \cdot g_i[1]$$

Similarly, the resultant condition for subtraction is $A_i^U + (\overline{B_i + 6})^U \geq 8$. Since $\overline{B_i + 6} = 15-(B_i+6)$, the condition is expressed as $A_i^U+(15-B_i)^U-6 \geq 8$, resulting in $A_i^U+(\overline{B_i})^U \geq 14$. As before, we define a control signal t_i which is true when this condition is verified. The value of t_i in terms of the binary carry–kill and carry–generate functions (using the bits of A_i and $\overline{B_i}$) results in:

$$t_i = a_i[3] + \overline{k}_i[3](g_i[2] + \overline{k}_i[2]\overline{k}_i[1]) \tag{5}$$

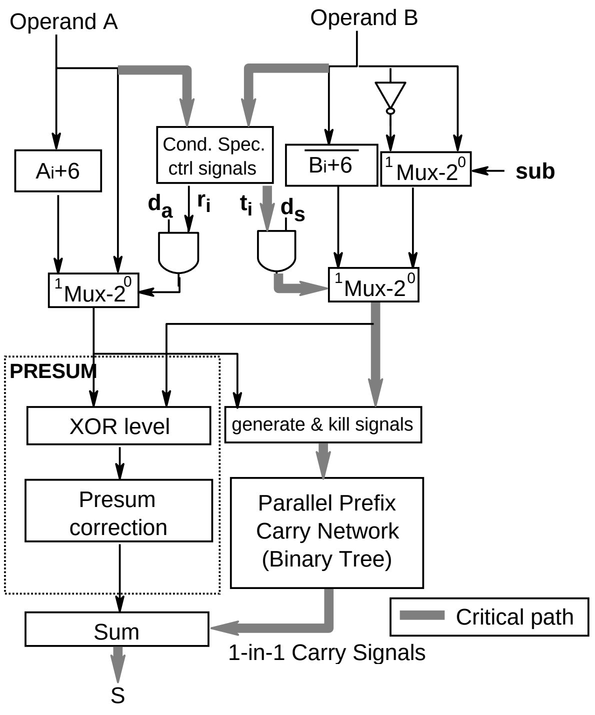
Therefore, the value of $(S_i^*)^U$ is determined in terms of r_i and t_i as follows ($d_a = 1$ for decimal addition and $d_s = 1$ for decimal subtraction):

$$(S_i^*)^U = \begin{cases} \bmod_{16}((A_i+6)^U + B_i^U + c_i[1]\cdot 2) & \texttt{If} \;\; d_a == 1 \text{ and } r_i == 1 \\ \bmod_{16}(A_i^U + B_i^U + c_i[1]\cdot 2) & \texttt{If} \;\; d_a == 1 \text{ and } r_i == 0 \\ \bmod_{16}(A_i^U + (\overline{B_i})^U + c_i[1]\cdot 2) & \texttt{If} \;\; d_s == 1 \text{ and } t_i == 1 \\ \bmod_{16}(A_i^U + (\overline{B_i+6})^U + c_i[1]\cdot 2) & \texttt{If} \;\; d_s == 1 \text{ and } t_i == 0 \end{cases}$$

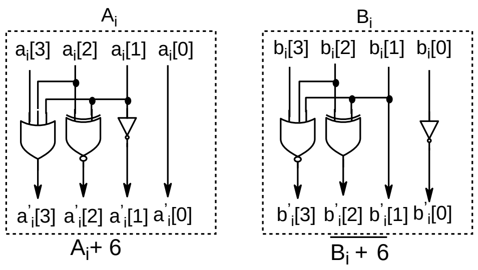
Figures 4 and 5 show two implementations of the algorithm. Figure 4(a) outlines the implementation of a binary/decimal adder for conditional speculative addition using a binary parallel prefix adder and additional hardware. For decimal addition, conditional speculation is performed using control signals $d_a \cdot r_i$ to select between $A_i + 6$ and A_i. The hardware implementation of operations $A_i + 6$ and $\overline{B_i + 6}$ is detailed in Figure 4(b). Signals $d_s \cdot t_i$ are used to control decimal speculative subtraction, selecting between $\overline{B_i + 6}$ and $\overline{B_i}$. Binary modes correspond to values of $d_a = d_s = 0$ while $sub = 1$ indicates a subtraction.

If the speculation is wrong, it is necessary to correct the resulting digits. Figure 4(c) shows a hardware implementation of the sum correction for a decimal digit. The sum correction scheme is placed after the XOR level in the pre–sum stage and is out of the critical path. The decimal sum correction scheme detects when $(S_i^*)^U = 14$ (or in binary $111-$) and corrects it. The value $(S_i^*)^U$ is 14 only when $p_i[3] = p_i[2] = p_i[1] = 1$ and $c_i[1] = 0$ (the $p_i[j]$ values are the pre–sum bits to obtain S_i^*). Therefore the post–correction can be done by evaluating $p_i[3] \cdot p_i[2] \cdot p_i[1]$ and replacing the value $111-$ by $100-$ using only a NAND–4 gate and two AND–2 gates, shown in black in Figure 4(c) (d indicates a decimal operation, that is, d_a or d_s enabled). Note that this decimal correction is not in the critical path (highlighted in Figure 4(c)), because the carry–in dependency is at the very last stage (XOR or MUX level) as in standard addition.

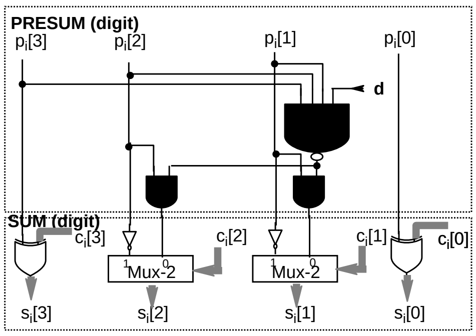
The proposed conditional speculative decimal addition/subtraction can also be implemented using a quaternary binary carry tree. In Figure 5(a) we show this implementation. To correct the speculative sum digits we use the modified 4–bit carry select adder of Figure 5(b). Additional gates for correction (shown off in black) are out of the critical carry path, so, as before, decimal correction does not increase the latency of the adder. To reduce the hardware complexity of the proposed quaternary–tree adder, we compute the control variables for conditional speculation as $r_i = a_i[3] + q_i$ and $t_i = \overline{a_i[3]} \cdot q_i$, where $q_i = b_i^*[3] + \overline{k_i[2]} + g_i[2]\overline{k_i[1]}$.



(a) Binary/Decimal Adder architecture

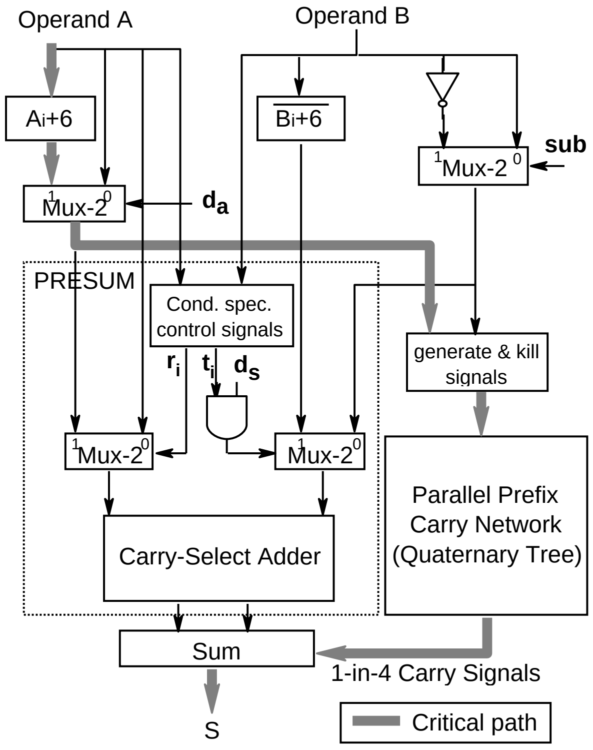


(b) Implementation of $A_i + 6$ and $\overline{B_i + 6}$

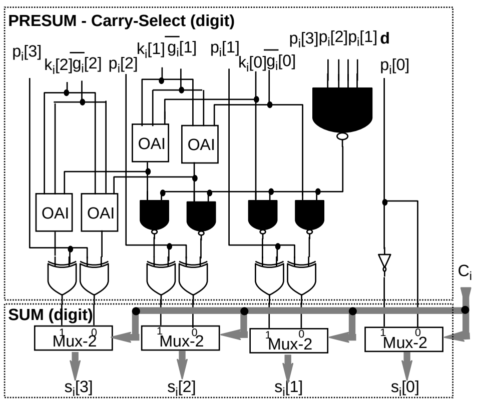


(c) Correction and final sum formation

Fig. 4. Proposed Combined Binary/Decimal Adder using a binary parallel prefix carry tree.



(a) Binary/Decimal Adder architecture



(b) Modified 4–bit carry select adder

Fig. 5. Proposed Binary/Decimal Adder using a quaternary carry tree.

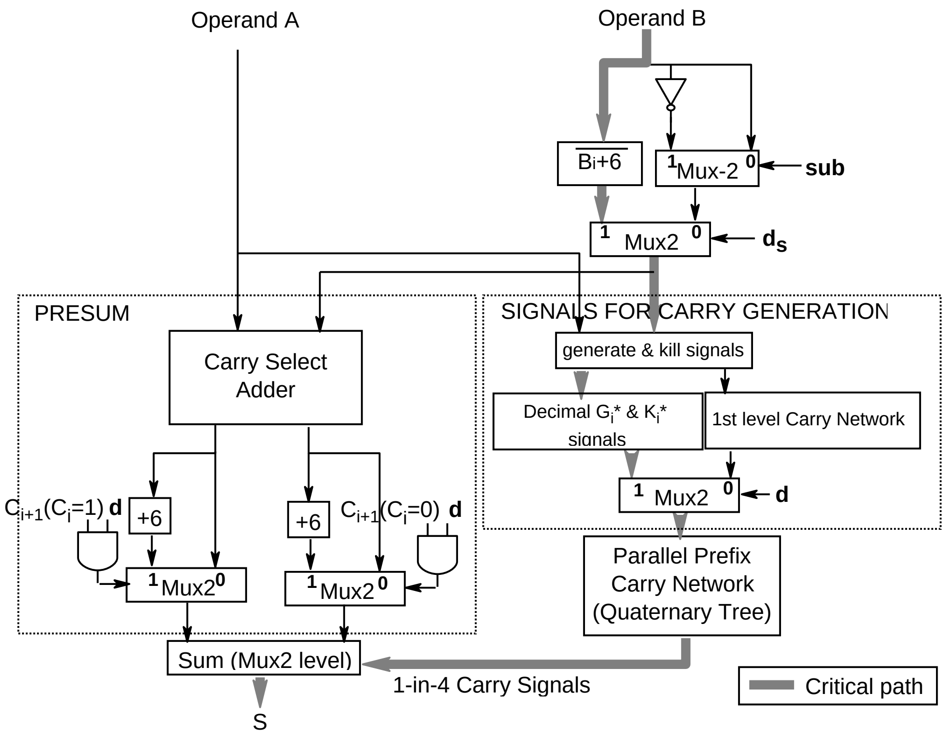


Fig. 6. Combined Binary/Direct Decimal Adder using a quaternary carry–tree.

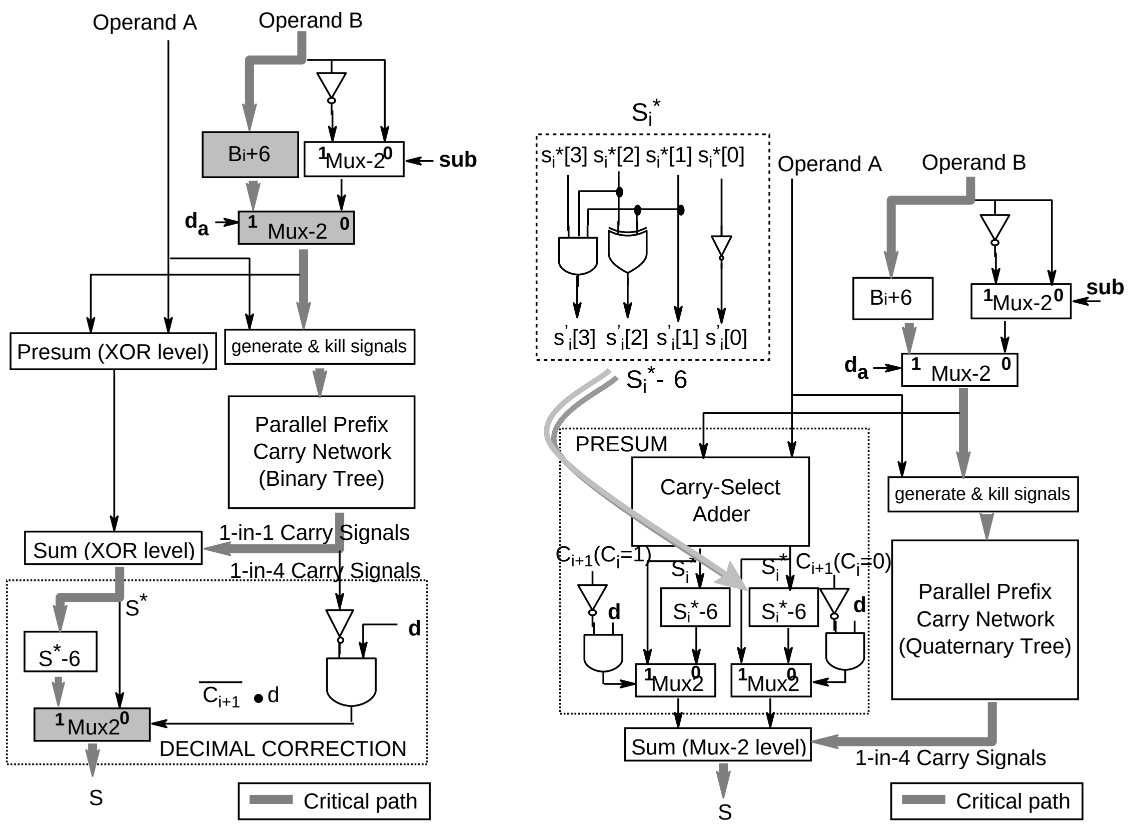
4 Delay–Area Estimations and Comparison

In this section we present a rough comparison among the different implementations using a delay model for static CMOS technology based on logical effort [11]. Using our model we have evaluated the delay and area of different binary, speculative, direct and conditional speculative (proposed in this work) decimal adders for three representative carry–tree topologies: Kogge–Stone (**K–S**), Ladner–Fischer (**L–F**) [8] and Quaternary Tree (**QT**) [9].

Figure 6 shows the architecture of an implementation of direct decimal addition/subtraction [4, 10]. We present the adder architecture with the following stages: operand setup, pre–sum, carry evaluation and sum. In the operand setup B_i^* is evaluated. Control signal d_s selects $\overline{B+6}$ for decimal subtraction. In the pre–sum stage, C_{i+1} and S_i^* are pre–evaluated for the two possible values of C_i using a 4–bit carry–select adder. The conditional C_{i+1} values are used to obtain $S_i(C_i = 0)$ and $S_i(C_i = 1)$ from $S_i^*(C_i = 0)$ and $S_i^*(C_i = 1)$ respectively, adding 6 to the sum digit when $C_{i+1} = 1$ as indicated by the algorithm. Finally, the decimal carries (C_i values), computed in the parallel prefix tree, select the correct sum digits in the sum stage.

The evaluation of G_i^* and K_i^* is performed using specific hardware for decimal operations while the evaluation of G_i and K_i is performed using the hardware of the binary carry–tree network. Binary carry–generate and carry–kill functions are combined in a first level of the carry–tree network not required for decimal addition. In this way, the remaining levels of the carry–tree network can be shared for decimal and binary operations. The control signal *sub* is enabled for subtraction while d is enabled for decimal operations.

Figure 7(a) shows a block diagram of a combined binary/decimal parallel–prefix adder using decimal speculative addition with a decimal post–correction scheme [12]. Binary and



(a) Binary prefix carry tree. (b) Quaternary carry tree.

Fig. 7. Combined Binary/Decimal Speculative Adder.

speculative decimal addition use the same carry network, sharing the same carry path. For decimal subtraction, the simplification $\overline{B_i + 6} + 6 = \overline{B_i}$ (with $\overline{B_i} \in [6, 15]$) is taken into account. Control signal d is enabled for decimal operations while d_a is only enabled for decimal addition. Post–correction is carried out using multiplexors to select $S_i = S_i^*$ when $C_{i+1} = 1$ or $S_i = S_i^* - 6$ when $C_{i+1} = 0$ (wrong speculation). For binary operations $S_i = S_i^*$.

To speed up speculative decimal addition, the evaluation and correction of sum digits may be performed in parallel to the carry computation as in direct decimal addition, using a 4–bit carry–select adder and a quaternary carry tree. Figure 7(b) shows this implementation [1–3, 6].

Table 1 presents the evaluation results for the three types of carry–tree topologies and the different decimal addition algorithms. As a reference we also show the results for the corresponding binary adders. The critical path delay for each adder is highlighted in the corresponding figure. To present the evaluation results we have considered the following stages: i) operand setup (hardware to prepare operands for addition or subtraction), ii) pre–sum (generation of carry–propagate signals and carry–select logic), iii) pre–carry (carry–generate and carry–kill signals are computed), iv) carry tree, v) sum (determination of final sum digits depending on carries by the use of XOR gates or 2:1 multiplexors) and vi) post–correction (hardware needed for decimal digit correction). For the comparison we use 64–bit operands. We also consider the same input and output loads for all the architectures in order to provide fair comparison results. Delay values are given in FO4 units (delay of an 1x inverter with a

		Binary		Direct Decimal [4, 10]		Speculative [1–3, 6, 12]		Proposed	
Prefix tree	Stage	Delay (t_{fo4})	Area (Nand2)	Delay (t_{fo4})	Area (Nand2)	Delay (t_{fo4})	Area (Nand2)	Delay (t_{fo4})	Area (Nand2)
K–S	Op. setup	2.50	240	–	–	3.80	520	5.95	1130
	Pre–sum*	2.00*	90	–	–	2.00*	90	3.90*	185
	Pre–carry	1.50	150	–	–	1.50	150	1.50	150
	Carry tree	7.40	960	–	–	7.40	960	7.40	955
	Sum	2.00	240	–	–	2.65	240	2.00	240
	Post–corr.	–	–	–	–	3.90	400	–	–
	Total	**13.40**	**1680**	–	–	**19.25**	**2360**	**16.85**	**2660**
L–F	Op. setup	2.50	240	–	–	3.80	520	5.95	1130
	Pre–sum*	2.00*	90	–	–	2.00*	90	3.90*	185
	Pre–carry	1.10	150	–	–	1.10	150	1.10	150
	Carry tree	9.20	585	–	–	9.20	585	9.20	585
	Sum	2.00	240	–	–	2.65	240	2.00	240
	Post–corr.	–	–	–	–	3.90	400	–	–
	Total	**14.80**	**1305**	–	–	**20.65**	**1985**	**18.25**	**2290**
QT	Op. setup	2.50	240	3.60	575	3.80	520	3.80	520
	Pre–sum*	3.75*	750	4.30*	1320	6.20*	1675	8.55*	1505
	Pre–carry	1.00	150	4.60	1020	1.00	150	1.00	150
	Carry tree	8.05	240	5.95	100	8.05	240	8.05	240
	Sum	2.70	240	2.70	240	2.70	240	2.70	240
	Total	**14.25**	**1620**	**16.85**	**3251**	**15.55**	**2825**	**15.55**	**2655**

* Not in the critical path.

Table 1. Delay and area estimations for 64–bit adders.

fanout of four 1x inverters) and the area values are in terms of the area of a minimum size NAND–2 gate. Since our aim is to compare different decimal addition methods and not to obtain precise absolute evaluation results, we assume gates with the drive strength of the minimum sized inverter (except buffers for high loads). Gate sizing to optimize the critical path and the effect of interconnections are beyond the scope of this rough comparison.

Figure 8 shows the delay–area space for the analyzed adders. From our comparison, we conclude that the direct decimal adder has no apparent advantage in comparison with the speculative and conditional speculative (proposed) with **Q–T** topology for carry generation (1.25 more hardware than our proposal and slightly more latency). For low latency the **Q–T** based schemes are the best choice. In this case our proposal requires slightly less hardware than the decimal speculative adder. For low hardware cost, the speculative adder with a **L–F** topology for carry generation fits the requirement (requires 0.75 times the hardware complexity of the proposed **Q–T** based scheme). For the case of trading–off hardware complexity and latency, the proposed **L–F** based scheme is a good alternative. Note that the **K–S** based schemes are not the best choice in any case, although they are close in terms of the estimated hardware complexity and latency. However, it is expected that real implementations with aggressive technologies and circuit techniques result in a significant degradation of its figures of merit due to its high routing complexity and the high amount of logic involved in critical paths.

Note that the fastest **Q–T** based decimal schemes are only 1.10 slower than the correspondent **Q–T** binary adder, although the hardware complexity increases by a factor of 1.65.

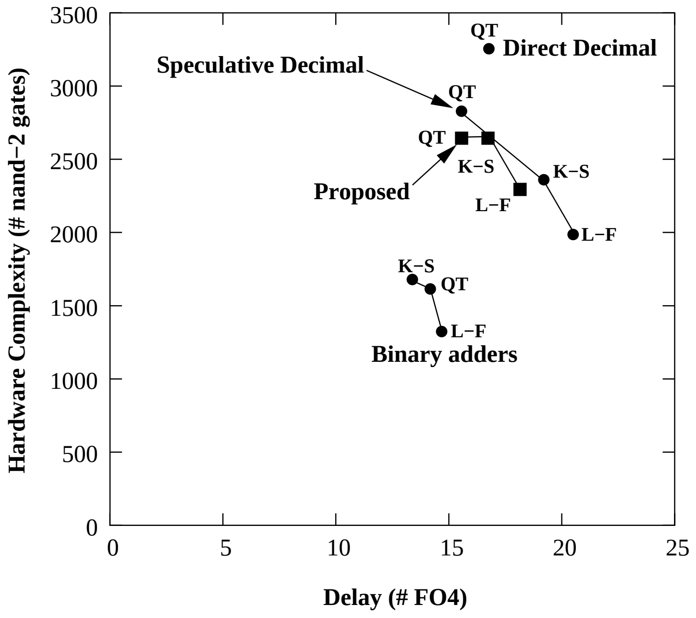


Fig. 8. Delay–area space.

5 Conclusions

In this work we present a new high–performance decimal addition/subtraction algorithm and its architecture that allows the computation of decimal or binary addition and subtraction operations. The algorithm proposed requires a minor post–correction so that any carry–tree topology can be used without the delay penalty of other schemes based on post–correction. The rough evaluations performed show that the proposed algorithm leads to adders with interesting area–delay figures. Moreover, the proposed algorithm might be of industrial interest since it is very competitive in comparison with commercial and patent–protected ones [1–4, 6]. Moreover, it opens new alternatives for exploring optimizations with aggressive circuit level techniques.

References

1. W. Bultmann, W. Haller, H. Wetter and A. Wörner. “Binary and decimal adder unit”. US Patent no US6292819, Sept. 2001.
2. F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz and S. R. Carlough. “The IBM z900 Decimal Arithmetic Unit”, Conference Record of the Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1335-1339, Nov. 2001.
3. F. Y. Busaba, T. Slegel, S. Carlough, C. Krygowski, J. G. Rell. “The design of the Fixed Point Unit for the z990 Microprocessor”, Proceedings of the 14th ACM Great Lakes Symposium on VLSI 2004, pp. 364-367, Apr. 2004.
4. M. A. Check, T. J. Slegel. “Custom S/390 G5 and G6 microprocessors”, IBM Journal of R&D, vol.43, no.5/6, pp. 671-680, Sept./Nov. 1999.
5. M. F. Cowlishaw, “Decimal Floating-Point: Algorism for Computers”, Proceedings of the 16^{th} IEEE Symposium on Computer Arithmetic, pp. 104-111, July 2003.
6. W. Haller, U. Krauch, T. Ludwig and H. Wetter. “Combined binary/decimal adder unit”. US Patent no : US 5928319, Jul. 1999.
7. IEEE Standards Committee, “Draft revision of the IEEE Standard for Floating–Point Arithmetic”. Available at `http://754r.ucbtest.org/drafts/754r.pdf`, May 2006.

8. S. Knowles. “A family of adders”, Proceedings of the 15th IEEE Symposium on Computer Arithmetic, pp. 277–284, June 2001.
9. S. K. Mathew, M. Anders, R. K. Krishnamurthy and S. Borkar. “A 4Ghz 130nm address generation unit with 32-bit sparse-tree adder core”, IEEE Journal of Solid-State Circuits, vol. 38, no. 5, pp. 689–695, May 2003.
10. M. Schmookler and A. Weinberger. “High Speed Decimal Addition”, IEEE Transactions on Computers, vol. c-20, no. 8, pp. 862-866, August 1971.
11. I. E. Sutherland, R. F. Sproull and D. Harris, “Logical Effort: Designing Fast CMOS Circuits”, Morgan Kaufmann, 1999.
12. J. Thompson, N. Karra, and M. J. Schulte. “A 64-bit Decimal Floating-Point Adder (extended version)”, Proceedings of the IEEE Computer Society Annual Symposium on VLSI, pp. 297-298, Feb. 2004.

Pipelined Architecture for Accurate Floating Point Range Reduction

Francisco J. Jaime, Javier Hormigo, Julio Villalba, and Emilio L. Zapata

Department of Computers Architecture
University of Málaga
{franj,hormigo,julio,ezapata}@ac.uma.es

Abstract. In this paper we present a novel pipelined architecture to deal with range reduction for floating point representation. It is based on the double residue method and a new system to generate elementary residues based on the previous ones. The generation of residues prevents from replicating the initial table for the pipelined implementation. To ensure an accuracy of one unit in the last place (ULP), the initial elementary residues have been enlarged. To reduce the number of stages, a radix-4 architecture is also proposed.

1 Introduction

Range reduction is a crucial step in the computation of elementary functions. A poor range reduction can lead to catastrophic accuracy problems when input arguments are large, as reported in [1]. Different software and hardware solutions have been proposed in the literature to deal with this problem [2], [3], [4], [5].

The only full hardware solution that solves the problem of dealing with irrational numbers (like π) and providing an accuracy of one unit in the last place (ULP) is presented in [6]. Nevertheless, it has a relatively low throughput due to the iterative nature of the implementation. In this paper we extend the double residue modular range reduction to the pipelined case in order to increase the throughput. This is not straightforward since at first it requires the replication of the elementary residues table (one table for each stage). Our approach avoids this table replication by applying a new algorithm which computes the elementary residues in every stage.

In section 2 we describe the basis of the double residue modular range reduction of [6]. Next, in section 3 we propose a method to generate elementary residues. In section 4 we propose a pipelined architecture for the floating point case. In section 5 we deal with precision errors, offering a solution to keep an accurate result by using guard bits. Section 6 explains how to improve the architecture using radix-4 codification and finally conclusions are discussed in section 7.

2 Double residue modular range reduction

The algorithm implemented is completely explained in [6], but here will be given an overall description, considering a floating point number similar to IEEE 754 standard for both the input argument (X) as well as the final reduced argument (R). Let X be a positive floating point number such that:

$$X = \mathrm{man}(X) \times 2^E = (\sum_{i=0}^{n-1} x_i 2^{-i}) \times 2^E \tag{1}$$

where $\text{man}(X)$ is the input vector mantissa, which has a length of n bits: $\text{man}(X) = (x_0 \,.\, x_1\, x_2 \ldots x_{n-1})$.

The aim is to quickly and accurately perform the operation:

$$R = (X - kA) \quad 0 \leq R < A \tag{2}$$

where A is a positive real number and R is the normalized floating point representation of the final reduced argument rounded by truncation.

The main idea of the algorithm itself is as follows:

1. The positive and negative elementary residues ($m_i^+ = 2^i \bmod A$ and $m_i^- = (2^i \bmod A) - A$) are stored into a look-up table. Both of them must always fulfill the following equation, which is the positive and negative residues definition itself:

$$m_i^+ - m_i^- = A \tag{3}$$

2. Starting from a value of zero, it goes on adding these elementary residues for each bit of the input vector mantissa ($\text{man}(X)$). The new residue obtained from the table is only considered when the corresponding bit from the input vector is set to 1, that is, the following equation is computed in each iteration:

$$R(i+1) = R(i) + x_i \cdot m_i^* \tag{4}$$

where $R(i+1)$ is the value accumulated through all the additions, $R(0) = 0$ and m_i^* is calculated as follows:

$$m_i^* = \begin{cases} m_i^+ & \text{if} \quad R(i) < 0 \\ m_i^- & \text{if} \quad R(i) \geq 0 \end{cases} \tag{5}$$

3. At the end, when the whole input vector X has been scanned, the last accumulated value $R(n)$ is checked: if it is a positive result, then $-A$ is added, otherwise the final result keeps the same as $R(n)$. This step is needed to ensure a final residue within the range $[-A, A)$ (see [6]).

Having a look at the iterative architecture detailed in [6], it can be seen that elementary residues m_i^+ and m_i^- are calculated in advance and stored into a look-up table. Then the circuit only has to access the corresponding look-up table entry and reads the right value. Equation (4) is performed by using a 3-2 carry save adder in an iterative process.

The most interesting case is that related to $A = 2\pi$ since it is applicable to the range reduction of the trigonometric functions. In this case, to achieve a precision of one ULP it is necessary to store the elementary residues with 54 bits of precision [6]. To avoid having a 54 bits data path, the table of elementary residues is chopped in two pieces, containing each one half of the bits of the elementary residues approximately. The architecture starts dealing with the first piece (which contains the most significative bits). If the precision obtained after the first pass is not enough, a second pass is required and the precision required by IEEE 754 is always achieved.

3 Generation of elementary residues

Our aim is to design a pipelined architecture for the double residue modular range reduction. At first, it requires one table for each stage of the pipeline (say "n"). This is due to the fact that the input argument may be any IEEE 754 representable number and the mantissa bits weight goes from -127 to 127 (from -21 to 127 if $A = 2\pi$ [6]). This solution supposes an unacceptable hardware cost, which is prevented with the algorithm that we propose in this section.

A pipelined architecture has to use the table in a different way (see section 4). Here m_i^+ and m_i^- are not loaded directly from memory, but calculated every time they are needed, following the next expression:

$$m_{i+1}^+ = \begin{cases} 2m_i^+ & \text{if} \quad 2m_i^+ < A \\ 2m_i^+ - A & \text{if} \quad 2m_i^+ \geq A \end{cases} \tag{6}$$

m_{i+1}^- is calculated according to the relationship between m_i^+ and m_i^-:

$$\text{If} \quad m_{i+1}^+ = 2m_i^+ \Longrightarrow m_{i+1}^- = 2m_i^- + A \tag{7}$$

$$\text{If} \quad m_{i+1}^+ = 2m_i^+ - A \Longrightarrow m_{i+1}^- = 2m_i^- \tag{8}$$

Relations (7) and (8) can be proved applying equation (3), which can also be written as follows:

$$m_i^+ = A + m_i^- \tag{9}$$

Proving relation (7):

$$\begin{array}{ll} m_{i+1}^+ = 2m_i^+ & \Longrightarrow A + m_{i+1}^- = 2(A + m_i^-) \Longrightarrow \\ A + m_{i+1}^- = 2A + 2m_i^- & \Longrightarrow m_{i+1}^- = 2m_i^- + A \end{array}$$

Proving relation (8):

$$\begin{array}{lll} m_{i+1}^+ = 2m_i^+ - A & \Longrightarrow A + m_{i+1}^- = 2(A + m_i^-) - A & \Longrightarrow \\ A + m_{i+1}^- = 2A + 2m_i^- - A & \Longrightarrow A + m_{i+1}^- = A + 2m_i^- & \Longrightarrow \\ m_{i+1}^- = 2m_i^- & & \end{array}$$

The piece of hardware in charge of computing these operations is called *NEXT m* and it can be seen in figure 1 b. It receives m_i^+ and m_i^- in carry save representation and $-A$ and A in conventional representation.This block is built using a CSA, getting a delay time suitable for the circuit.

Making the comparisons required by equation (6) has a high cost. In order to overcome this problem a flags vector has to be precalculated and stored into the look-up table before the circuit starts to work. This vector contains a flag for every *NEXT m* block, thus each step within the architecture will retrieve one flag from this vector and will pass the rest to the next step.

Each flag controls the way m_{i+1}^+ and m_{i+1}^- are computed following expressions (6), (7) and (8):

Flag=0 : $m_{i+1}^+ = 2m_i^+$ and $m_{i+1}^- = 2m_i^- + A$
Flag=1 : $m_{i+1}^+ = 2m_i^+ - A$ and $m_{i+1}^- = 2m_i^-$

NEXT m block at step 1 (called *NEXT* m*) is different from the rest (see figure 1 a) because it receives m_i in conventional representation, whereas the others only work with carry save representation. *NEXT* m* could have been like *NEXT m*, but this way some hardware is saved. *NEXT* m* is also faster than *NEXT m*, however this is not very important since the delay time for every step will be determined by the slowest one.

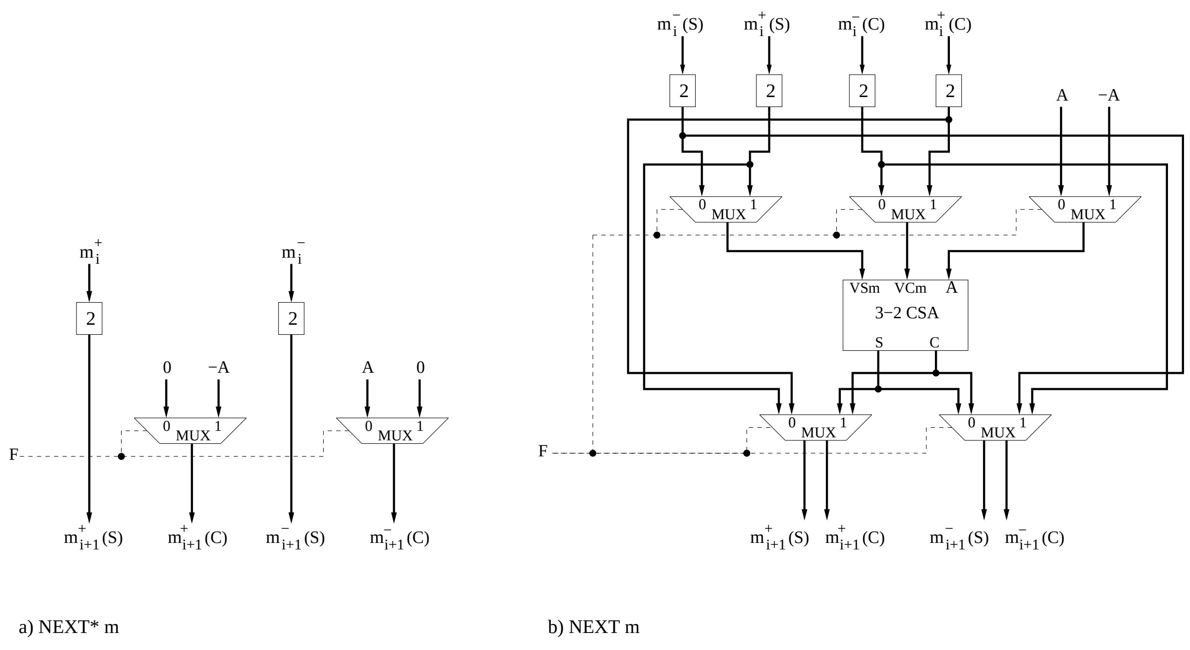


Fig. 1. Hardware for *NEXT m*.

4 Architecture for floating point representation

The proposed architecture is presented in figure 2. It can be seen a look-up table which stores every positive and negative residues for powers of 2 mod A which could be needed by the circuit first step, that is, considering all exponent possible values (from the input vector X). These residues have to be calculated in advance, together with the flags vectors described in section 3.

Every step i carries out two main functions at the same time: calculating (m^+_{i+1}, m^-_{i+1}) and the accumulative addition described in equation (4), that is, $R(i)$. A general step i and step $n-1$ can be seen in figure 3.

Initially, the exponent within X addresses the proper couple of residues and the corresponding flags vector from the table. This data will be used by step 1 to simultaneously compute (m^+_2, m^-_2) and the first accumulated sum value $R(1)$ (see equation (4)) following the methods described in sections 2 and 3. When step 1 finishes, $R(1)$ and (m^+_2, m^-_2) pass to step 2 and the circuit keeps going on until step n is reached. This step uses the last pair of elementary residues, produced by step $n-1$, and hence it doesn't calculate any new one, only computing the accumulative value $R(n)$, which will go through the last step (step $n+1$), in charge of adding $-A$ when necessary in order to ensure a valid final result within the range $[-A, A)$. The result obtained comes in carry save representation.

5 Precision errors boundary

Although a final error is unavoidable, the ideal result would be produced if the circuit could work with exact precision in every operation and only had to truncate the final value at the end to get a floating point representation. In this case a minimum error of one ULP is achieved. In this section we study the precision error propagation and the minimum number of guard bits that the circuit requires to obtain a final precision of one ULP.

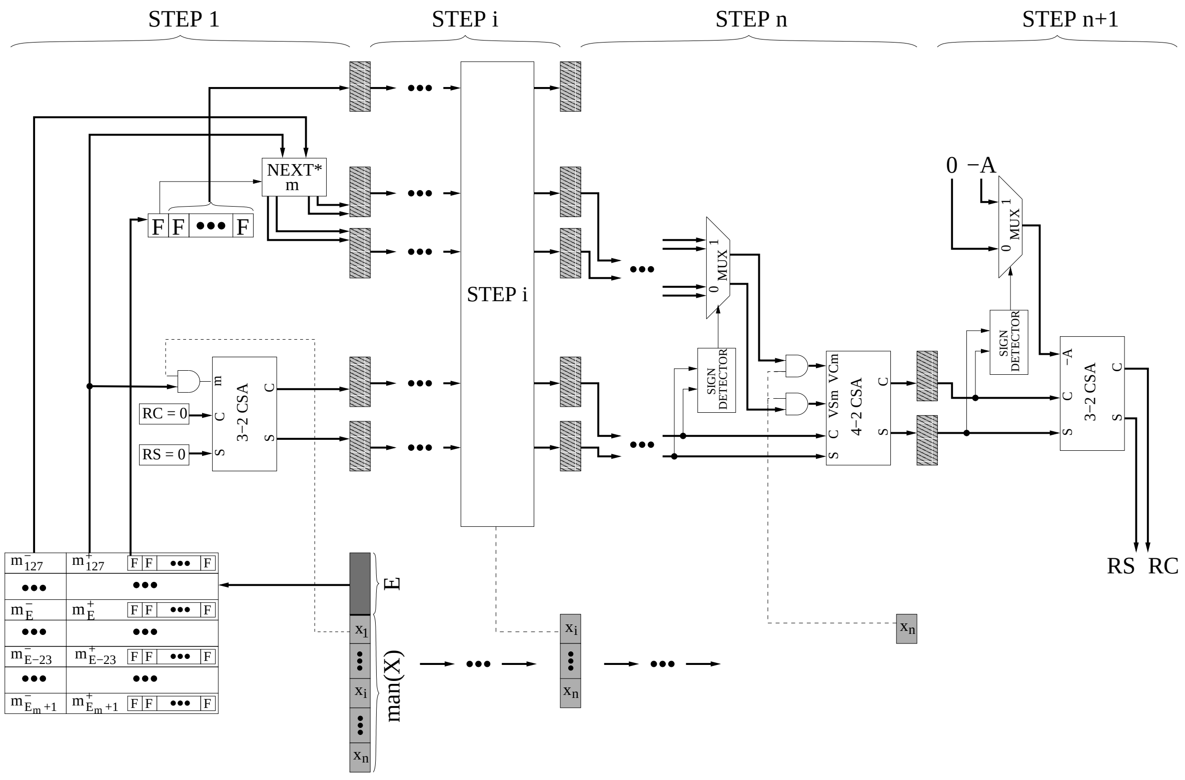


Fig. 2. Pipelined architecture for floating point representation.

There are two different sources for precision errors propagation inherent to the pipelined circuit described above (see figure 2):

1. Error propagation when computing the accumulative addition $R(i)$.
2. Error propagation when computing m^+_{i+1} and m^-_{i+1}, that is, when *NEXT m* block is working.

In order to study the error propagation, a real number R is going to be considered as $R = v + e$, where v is the floating point representation value and e is the error due to truncation.

5.1 Error propagation computing $R(i)$

Let's consider a real number with t fractional bits used by the floating point representation plus g guard bits:

$$\underbrace{x_1\ x_2\ x_3\ x_4\ \ldots\ x_t}_{t \text{ bits for representation}}\ \underbrace{x_{t+1}\ x_{t+2}\ \ldots\ x_{t+g}}_{g \text{ guard bits}}$$

Thus, for our target precision the value of one ULP is delimited by the following expression:

$$ULP < 2^{-t} \tag{10}$$

If guard bits are considered, then the new error ge is delimited by:

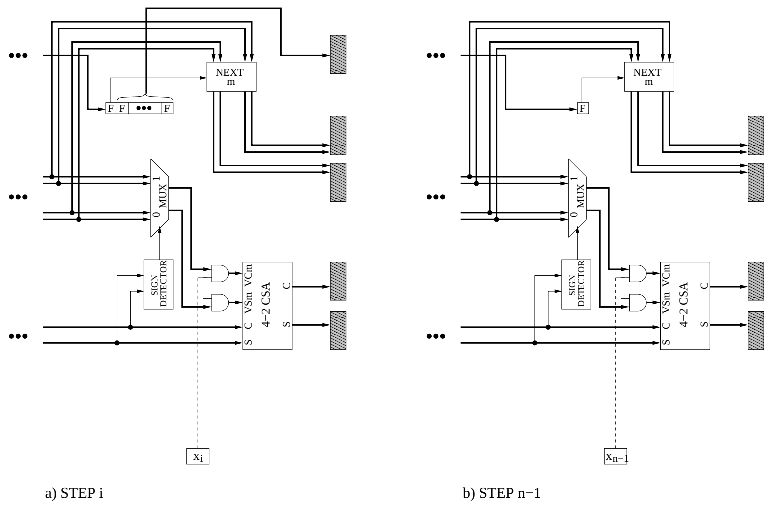


Fig. 3. General step and step $n-1$.

$$ge < 2^{-t-g} \tag{11}$$

Considering numbers represented with $t+g$ fractional bits, every time an accumulative addition is performed (to compute $R(i)$), the precision error increases:

$$\begin{array}{ll} R_1 + R_2 = v_1 + e_1 + v_2 + e_2 = v_3 + e_3 = R_3 & e_3 < 2ge \\ R_3 + R_4 = v_3 + e_3 + v_4 + e_4 = v_5 + e_5 = R_5 & e_5 < 3ge \end{array}$$

As it can be seen, the error is increased by one ge in every addition, thus after h consecutive additions, the final error will be delimited by $h \cdot ge$. Therefore, in order to keep a result with an error no bigger than one ULP, expression (12) must be fulfilled:

$$h \cdot 2^{-t-g} < 2^{-t} \tag{12}$$

The number of guard bits g needed can be retrieved from equation (12):

$$g > \log_2 h \tag{13}$$

Addind $g = \lceil \log_2 (h+1) \rceil$ guard bits to the operands used to compute $R(i)$, after h consecutive additions, it can be ensured an $R(h)$ value with a precision error no bigger than one ULP.

Let's see a short example of the guard bits use by making four additions. We will consider a machine number with $t = 14$ fractional bits, and hence that is the desired precision. However, the exact real numbers precision is 23 bits. Because four additions are going to be made, then $g = \lceil \log_2(4+1) \rceil = 3$ guard bits are used.

Considering the exact number $X_e = 1.10011001000101110000101$, the performed operation is $R_e = X_e + X_e + X_e + X_e + X_e$, which can be seen in Table 1.

The first $t = 14$ fractional bits of the result is the correct value we should get when using a machine representation with such an amount of bits. Nevertheless, error propagation due to truncation provides a different result, as shown in Table 2, where the 3 least significant bits are corrupted.

$t = 14$ fractional bits	9 bits
1.10 011 001 000 101	110 000 101
+ 1.10 011 001 000 101	110 000 101
11.00 110 010 001 011	100 001 010
+ 1.10 011 001 000 101	110 000 101
100.11 001 011 010 001	010 001 111
+ 1.10 011 001 000 101	110 000 101
110.01 100 100 010 111	000 010 100
+ 1.10 011 001 000 101	110 000 101
111.11 111 101 011 100	110 011 001

Table 1. $R_e = 5X_e$ computation. $X_e = 1.10011001000101110000101$

$t = 14$ fractional bits
1.10 011 001 000 101
+ 1.10 011 001 000 101
11.00 110 010 001 010
+ 1.10 011 001 000 101
100.11 001 011 001 111
+ 1.10 011 001 000 101
110.01 100 100 010 100
+ 1.10 011 001 000 101
111.11 111 101 011 001

Table 2. $R_r = 5X_r$ computation. $X_r = 1.10011001000101$

If operands with $g = 3$ extra guard bits are considered, we can achieve the requested accuracy, see Table 3, where the first 14 fractional bits of the result are accurate.

$t = 14$ fractional bits	$g = 3$
1.10 011 001 000 101	110
+ 1.10 011 001 000 101	110
11.00 110 010 001 011	100
+ 1.10 011 001 000 101	110
100.11 001 011 010 001	010
+ 1.10 011 001 000 101	110
110.01 100 100 010 111	000
+ 1.10 011 001 000 101	110
111.11 111 101 011 100	110

Table 3. $R_g = 5X_g$ computation. $X_g = 1.10011001000101110$

In the pipelined architecture shown in figure 2, the number of additions depends on the input vector length. Considering an input vector with n bits, then $n+1$ additions are performed. Therefore this particular circuit will need $g = \lceil \log_2 (n+2) \rceil$ guard bits.

5.2 Error propagation computing m^+_{i+1} and m^-_{i+1}

m^+_{i+1} and m^-_{i+1} values computed by *NEXT m* are consumed by the CSA in charge of computing $R(i)$. As it has been said in section 5.1, this CSA needs its operands with a length of $t+g$ in order to keep a precision with an error less than one ULP, where t is the number of bits used by the floating point representation and g the number of guard bits in the R(i) data path. Thus, in this case, *NEXT m* blocks will have to provide values with exact precision until bit $t+g$.

$$\underbrace{y_1\ y_2\ y_3\ y_4\ \cdots\ y_t}_{t \text{ bits for representation}}\ \underbrace{\overbrace{y_{t+1}\ y_{t+2}\ \cdots\ y_{t+g}}^{g \text{ extra precision bits}}\ y_{t+g+1}\ y_{t+g+2}\ \cdots\ y_{t+j}}_{j \text{ guard bits}}$$

Considering a *NEXT m* block with j guard bits operands, the precision error je inherent to each operand is delimited by equation (14):

$$je < 2^{-t-j} \tag{14}$$

Similarly to what happens in section 5.1, every time a *NEXT m* block operates the error increases. Watching figure 1 it can be seen how every block makes a multiplication by 2 and an addition. Let's see how an error propagates through these operations:

$$\begin{aligned} 2R_1 + R_2 = 2v_1 + 2e_1 + v_2 + e_2 = v_3 + e_3 = R_3 \quad e_3 < 3je \\ 2R_3 + R_4 = 2v_3 + 2e_3 + v_4 + e_4 = v_5 + e_5 = R_5 \quad e_5 < 7je \end{aligned}$$

Here appears a run of numbers following a recursive equation:

$$\begin{cases} a_0 = je \\ a_k = 2a_{k-1} + je \end{cases} \tag{15}$$

Equation (15) is a linear recursive equation with constant coefficients, thus using its initial conditions and its characteristic equation, it can be transformed into a non recursive equation:

$$a_n = je(2^{k+1} - 1) \tag{16}$$

Better said, $je(2^{k+1}-1)$ is the final propagated error after k *NEXT m* blocks have operated. Hence expression (17) must also be fulfilled:

$$2^{-t-j}(2^{k+1} - 1) < 2^{-t-g} \tag{17}$$

Making some calculations with expression (17), the number of guard bits j needed can be solved:

$$j > g + \log_2 (2^{k+1} - 1) \tag{18}$$

Analogously to section 5.1, the number k of *NEXT m* blocks depends on the length of the input vector. Considering an input vector with n bits, then $n - 1$ blocks are used, and hence the number of guard bits is $j = g + n > g + \log_2 (2^n - 1)$.

As conclusion of this section, $g = \lceil \log_2 (n + 2) \rceil$ guard bits are required for the $R(i)$ data path and $j = n + \lceil \log_2 (n + 2) \rceil$ guard bits are required for the m_i data path.

6 Radix-4 coding

Radix-4 coding can be applied to the input vector in order to reduce the number of steps for the pipelined architecture, getting a considerable hardware cost reduction.

When using Radix-4 coding, the values of the input vector X will change from $x_i \in \{1, 0\}$ to $x_i \in \{-2, -1, 0, 1, 2\}$. After applying this coding, equation (4) is changed into equation (19):

$$R(i + 1) = R(i) + \sigma_i m_i^* \tag{19}$$

Where $i = 0, 2, 4, \ldots$ and $\sigma_i = \text{sign}(x_i)$. In this case, m_i^* follows equation (20) instead of equation (5):

$$m_i^* = \begin{cases} m_{i+1}^+ & \text{if} \quad R(i) < 0 \quad \text{and} \quad |x_i| = 2 \\ m_i^+ & \text{if} \quad R(i) < 0 \quad \text{and} \quad |x_i| = 1 \\ 0 & \text{if} \quad |x_i| = 0 \\ m_i^- & \text{if} \quad R(i) \geq 0 \quad \text{and} \quad |x_i| = 1 \\ m_{i+1}^- & \text{if} \quad R(i) \geq 0 \quad \text{and} \quad |x_i| = 2 \end{cases} \tag{20}$$

Using Radix-4 coding, each step analyzes two input bits instead of only one. With this technique the circuit area and latency is smaller because only 50% of steps is needed. Nevertheless, due to coefficient σ_i in equation (19), every step must be capable of making a subtraction, apart from making an addition, and thus, the cycle time increases.

On the other hand, some extra time is also needed to recode the input vector into Radix-4 coding, but this can be done using one or more steps at the beginning of the circuit.

7 Conclusion

In this paper we have presented a novel pipelined carry save architecture to deal with the range reduction for elementary functions (trigonometric and exponential), based on the double residue method proposed in [6] for the iterative design. The architecture is intended for floating point representation, which is specially useful for the case of 2π since it allows us to have precision of one ULP for any input argument. The delay of each stage is similar to the cycle time of the iterative design, increasing the throughput to one computation per cycle. To reduce the latency of the system, a radix-4 alternative is also proposed.

Due to pipeline nature, its hardware cost is higher than the iterative version one. For an architecture working with IEEE 754 single precision input arguments and $A = 2\pi$, it requires: a 3,4 KB look-up table, 2828 FAs, 45 2-1 AND gates, 130 2-1 multiplexers, 23 3 bits CLAs, 46 35 bits registers, 44 58 bits registers and a few more registers to store the flags vector within each stage.

We have proposed a specific hardware that permits the generation of the next elementary residue from the previous one, preventing the replication of the table, which keeps the same size than that within the iterative design [6]. We have also provided a study of the error so that the width of the elementary residues stored in the table has been enlarged, ensuring a precision of one ULP for all cases.

References

1. J. M. Muller. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997.
2. N. Brisebarre, D. Defour, P. Kornerup, J. M. Muller, and N. Revol. A new range-reduction algorithm. *IEEE Transactions on Computers*, 54:331–339, March 2005.
3. W. Cody and W. Waite. *Software Manual for the Elementary Functions.* Prentice-Hall, Englewood Cliffs, N. J., 1980.
4. M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *J. Universal Computer Science*, 1:162–175, March 1995.
5. M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.
6. Julio Villalba, Tomas Lang, and Mario A. González. Double-residue modular range reduction for floating-point hardware implementations. *IEEE Transactions on Computers*, 55:254–267, March 2006.

7. Julio Villalba. *Diseño de arquitecturas CORDIC multidimensionales.* Ph.d. thesis, University of Málaga, 1995.
8. Milos D. Ercegovac and Tomas Lang. *Digital Arithmetic.* Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 2004.

Decidability of Collision between a Helical Motion and an Algebraic Motion

Sung Woo Choi[1], Sung-il Pae[2], Hyungju Park[2], and Chee K. Yap[3]

[1] Duksung Women's University
Seoul, Korea
[2] Korea Institute for Advanced Study
Seoul, Korea
[3] Courant Institute, New York University
New York, NY USA

Abstract. We show in this paper that collision between two moving balls in $3D$ is decidable if one of the bodies has a helical motion and the other body has an algebraic motion. Furthermore, an explicit polynomial time complexity bound is derived for this problem. Such bounds depend on effective versions of Baker's theorem on linear form in logarithms in transcendental number theory.

1 Introduction

Many geometric problems are solved using the Real RAM model. As long as the solutions remain algebraic, the use of a Real RAM model is feasible. But when transcendental operations such as $\sin x$ and $\exp x$ are involved, it is a major open problem in Exact Geometric Computation [10] whether the Real RAM model can be simulated by a Turing machine. Recently, the first example of a transcendental geometric problem that is provably solvable in the Turing machine model was shown in [2]: this is the problem of computing shortest paths amidst disc obstacles.

In this paper, we study a collision detection problem that is also transcendental in nature. It is well-known that algebraic motion planning is solvable since the early 1980s [6]. Here, obstacles are typically static and some feasible motion is to be computed. Superficially, such motions may involve the trigonometric functions such as $\sin x$ and $\cos x$; but they can be resolved by introducing algebraic relations such as $\sin^2 x + \cos^2 x = 1$.

But truly transcendental motion planning problems can arise through the introduction of helical motions. In modeling, computer graphics and robotics, the use of helical motions or geometry is relatively common (e.g., [7, Section 3.1.3], [3, 5]). The simplest helical motion is that of a point p that is moving along a fixed direction u at constant velocity while simultaneously rotating about a fixed axis that is along direction u. Let us suppose that simultaneously, a body $\mathcal{B}$ is moving in some known motion. We want to decide if p and $\mathcal{B}$ will collide. In this paper, we will give decision procedures for answering such questions. Our procedures will be shown to be implementable on Turing machines, not just Real RAMs. As in [2], such results will depend on zero bounds from transcendental number theory.

The possibility of computing such potential collision may seem to be of purely theoretical interest. Nevertheless, there may be need practical need for very high accuracy computations of this sort. On July 4th 2005, in the dramatic display of precision engineering and calculations, NASA successfully sent a man-made projectile into collision course with the comet Tempel 1, traveling with a relative speed of 23,000 mph. More generally, we want to predict if two celestial bodies will ever collide – this may involve computing very far into the future, with guaranteed accuracy. Presumably such questions will arise in the future.

Instead of asking if two bodies will collide, we can also ask if two bodies will come within distance $\varepsilon \geq 0$ of each other. This near-collision problem is a slight generalization of the collision question. For algebraic bodies, their decidability is equivalent. As we shall see, our ability to answer such questions is fairly limited.

Contributions of this paper. Helical motions are one of the simplest form of non-algebraic motion that are used in applications. This paper shows that a collision problem involving such motion is computable. This adds to our currently sparse collection of transcendental geometric problems known to be computable. The boundary between what is and what is not computable "exactly" in the geometric sense is a fundamental issue in the theory of real computation, and of complexity theory. This area also has practical implications for robust geometric computations.

Overview of Paper. In Section 2, we introduce a simple version of the collision problem involving helical motion. This problem is shown to be decidable. In Section 3, we derive a polynomial time complexity bound assuming the input motions are defined by rational polynomials. In Section 4, we discuss possible extensions.

2 A Simple Case

Let p be a point in a helical motion and let $h(t) = (\cos t, \sin t, st)$ be the position of p at time t. Suppose that $\mathcal{B}$ is a ball with radius r that moves along a curve and let $c(t) = (c_1(t), c_2(t), c_3(t))$ be the position of $\mathcal{B}$'s center at time t.

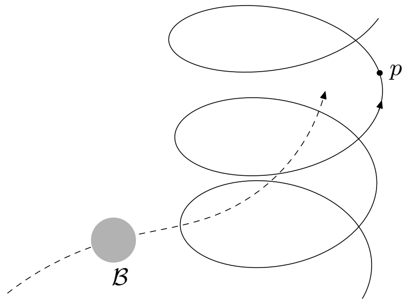


Fig. 1. Collision detection of a point p in helical motion and a moving ball $\mathcal{B}$.

Assuming that p and $\mathcal{B}$ are not in collision at the initial time $t = 0$, we want to decide whether p and $\mathcal{B}$ will ever collide at a time $t > 0$. Assume that $c_i(t)$'s are algebraic functions, *i.e.*, there exists a polynomial $P(x, y) \in \mathbb{C}[x, y]$ such that $P(c_i(t), t) \equiv 0$. We shall focus on an interval $I = [T_1, T_2]$ on which $c(t)$ is continuous and differentiable (except at the boundaries), where T_1 and T_2 are algebraic numbers. The most important class of functions in practice for $c_i(t)$'s would be polynomials with rational coefficients, and we will focus on this case in Section 3. Another interesting class is piecewise continuous rational functions, and the assumption on $[T_1, T_2]$ is natural in this situation. In this section, we assume that $c_i(t)$'s are general algebraic functions, and the radius r of $\mathcal{B}$ and the speed parameter s of $h(t)$ are real algebraic numbers. We show that the corresponding collision detection is decidable.

There is a collision if and only if the inequality

$$\|h(t) - c(t)\| \le r \tag{1}$$

has a real solution in $t \in I$. The inequality (1) is equivalent to

$$\|h(t) - c(t)\|^2 = (\cos t - c_1(t))^2 + (\sin t - c_2(t))^2 + (st - c_3(t))^2 \le r^2.$$

By continuity, this is equivalent to checking if there is a solution for the equation $\|h(t) - c(t)\|^2 = r^2$ for some $t \in I$. This equation is now equivalent to checking the solvability of equation of the form

$$a(t)\cos t + b(t)\sin t + d(t) = 0, \tag{2}$$

where $a(t) = -2c_1(t)$, $b(t) = -2c_2(t)$, and $d(t) = c_1(t)^2 + c_2(t)^2 + (st - c_3(t))^2 + 1 - r^2$. Note that $a(t)$, $b(t)$, and $d(t)$ are differentiable on (T_1, T_2) and continuous on $[T_1, T_2]$.

Theorem 1. *It is decidable whether there is a real solution of the equation of type (2).*

Note that equation (2) includes transcendental functions. To prove the theorem, we transform the equation into a form to which we can apply the zero bound. (See Section 3.2.) Let $A_0 = \{t \in I \mid a(t)^2 + b(t)^2 > 0\}$, the set of t for which $a(t)$ and $b(t)$ are not simultaneously zero. For $t \in A_0$, let

$$\alpha(t) = \frac{a(t)}{\sqrt{a(t)^2 + b(t)^2}}, \quad \beta(t) = \frac{-b(t)}{\sqrt{a(t)^2 + b(t)^2}}, \quad \delta(t) = \frac{-d(t)}{\sqrt{a(t)^2 + b(t)^2}}.$$

Then, for each $t \in A_0$, there is $\theta(t)$ such that $\cos\theta(t) = \alpha(t)$ and $\sin\theta(t) = \beta(t)$, and equation (2) is reduced to

$$\cos(t + \theta(t)) = \delta(t). \tag{3}$$

Let us fix a branch of arc cosine, say $\arccos : [-1, 1] \to [0, \pi]$. Then $\theta(t) = \arccos\alpha(t)$ when $\beta(t) \ge 0$, and $\theta(t) = -\arccos\alpha(t)$ when $\beta(t) \le 0$. Rewrite (3) as

$$\cos(t \pm \arccos\alpha(t)) = \delta(t), \tag{4}$$

subject to the sign of $\beta(t)$. Now let $A = \{\, t \in I \mid -1 \le \delta(t) \le 1,\ a(t)^2 + b(t)^2 > 0 \,\}$. By definition, we have $A \subseteq A_0$. Since $-1 \le \delta(t) \le 1$ on A, we can take arccos on both sides of (4) and obtain the following equations:

$$t + \arccos\alpha(t) - \arccos\delta(t) = 0 \mod 2\pi, \tag{5}$$
$$t + \arccos\alpha(t) + \arccos\delta(t) = 0 \mod 2\pi, \tag{6}$$
$$t - \arccos\alpha(t) - \arccos\delta(t) = 0 \mod 2\pi, \tag{7}$$
$$t - \arccos\alpha(t) + \arccos\delta(t) = 0 \mod 2\pi, \tag{8}$$

where equations (5) and (6) are subject to the condition $\beta(t) \ge 0$, and equations (7) and (8) are subject to the condition $\beta(t) \le 0$. The following lemma justifies the reduction of equation (2) to equations (5)–(8).

Lemma 2. *For $t \in A$, equation (2) holds if and only if one of the equations (5)–(8) holds.*

Proof. Suppose that t_0 is in A and satisfies (2). There is a unique $\theta_0 \in [0, 2\pi)$ such that $\cos\theta_0 = \alpha(t_0)$, $\sin\theta_0 = \beta(t_0)$. This θ_0 clearly satisfies $\cos(t_0 + \theta_0) = \delta(t_0)$. Since $t_0 \in A$, we have $-1 \leq \delta(t_0) \leq 1$, and thus

$$\arccos\delta(t_0) = \arccos\cos(t_0 + \theta_0) = \pm(t_0 + \theta_0) \pmod{2\pi}, \tag{9}$$

depending on which quadrant $t_0 + \theta_0$ is contained in. Now, depending on the sign of $\beta(t_0)$, we have either $\theta_0 = \arccos\alpha(t_0)$ or $\theta_0 = 2\pi - \arccos\alpha(t_0)$. By eliminating θ_0 in (9) with this, we conclude that t_0 satisfies one of (5)–(8).

Conversely, if t_0 satisfies equations (5) or (6), then clearly we have $\cos(t_0 + \arccos\alpha(t_0)) = \delta(t_0)$. Note that if $\beta(t_0) \geq 0$, then $\sin\arccos\alpha(t_0) = \sin t_0$. By expanding cosine, we obtain (2) for t_0. For equations (7) or (8), we have $\cos(t_0 + \arccos\alpha(t_0)) = \delta(t_0)$. Because $\beta(t_0) \leq 0$ in this case, then $\sin\arccos\alpha(t_0) = -\sin t_0$. So we obtain (2) for t_0 by expanding cosine. □

Lemma 2 takes care of the case where t is in A. If $t \in I - A$, the only case that our argument does not capture is when $a(t) = b(t) = 0$. The following lemma is clearly true.

Lemma 3. *If $t \in I - A$, equation (2) holds if and only if $a(t) = b(t) = d(t) = 0$.*

Now we show how to decide the existence of a zero of equation (2). Given algebraic functions $a(t)$, $b(t)$ and $d(t)$, we first check if $a(t), b(t)$ and $d(t)$ have a simultaneous zero in $[T_1, T_2]$, and this is clearly a decidable problem. If they have a solution, we can stop and conclude that there is a collision. In this degenerate case, $a(t) = -2c_1(t) = 0$, and $b(t) = -2c_2(t) = 0$. So the ball's center is at the axis of the helix. And $d(t) = (st - c_3(t))^2 + 1 - r^2 = 0$ means that p is touching the ball's surface. This can happen only when $r \geq 1$.

Now suppose that there is no simultaneous zero of $a(t), b(t)$ and $d(t)$ in $[T_1, T_2]$. By Lemma 3, we only need to check whether there is a zero in A, and the rest of this section is devoted to this question. We first show that A is a union of a finite number of closed intervals with algebraic endpoints.

Lemma 4. *If there is no simultaneous zero of $a(t), b(t)$ and $d(t)$ in $[T_1, T_2]$, then A is a union of a finite number of closed intervals with algebraic endpoints.*

Proof. Since there is no simultaneous zero of $a(t), b(t)$ and $d(t)$, if $a(t)^2 + b(t)^2 = 0$, then $d(t) \neq 0$. Because $d(t)$ does not vanish, by continuity of $a(t)$, $b(t)$ and $d(t)$, the condition $-1 \leq \delta(t) \leq 1$ implies that $a(t)^2 + b(t)^2 > 0$; otherwise, $\delta(t) = -d(t)/\sqrt{a(t)^2 + b(t)^2}$ would not be bounded in the neighborhood of zeros of $a(t)^2 + b(t)^2$. Hence, we can remove the condition $a(t)^2 + b(t)^2 > 0$ in the definition of A, and the set A is determined by only the inequality $d(t)^2 \leq a(t)^2 + b(t)^2$. The function $f(t) = d(t)^2 - a(t)^2 - b(t)^2$ is continuous on $I = [T_1, T_2]$, which is a compact set. The set $A = \{\, t \in I \mid d(t)^2 - a(t)^2 - b(t)^2 \leq 0 \,\} = f^{-1}((-\infty, 0]) \cap [T_1, T_2]$ is an intersection of a closed set and a compact set in $\mathbb{R}$ and therefore is a compact set. So A is a union of a finite number of closed intervals. The endpoints of the intervals in A are the zeros of f, T_1 or T_2, which are algebraic. □

Now we focus on each connected interval of A and decide whether equations (5)–(8) have a zero using a zero bound for linear forms in arc cosines. Consider one connected interval $[t_1, t_2] \subset A$ and equation (5). Note, again, that t_1 and t_2 are zeros of $\delta(t) \pm 1$, T_1 or T_2. For $t \in [t_1, t_2]$,

$$t_1 - 2\pi \leq t + \arccos\alpha(t) - \arccos\delta(t) \leq t_2 + 2\pi.$$

Therefore, we need to check only a finite number of equations

$$t + \arccos \alpha(t) - \arccos \delta(t) - 2n\pi = 0,$$

for integers n. For a fixed integer k, define $F : [t_1, t_2] \to \mathbb{R}$ by

$$F(t) = t + \arccos \alpha(t) - \arccos \delta(t) - 2k\pi,$$

and we are to check the existence of a zero of F in $[t_1, t_2]$. Note that F is continuous on $[t_1, t_2]$, but may not be differentiable on $[t_1, t_2]$ since arccos is not differentiable at ± 1. Since $\delta(t) \neq \pm 1$ on (t_1, t_2), we find the t values in (t_1, t_2) where $\alpha(t) = \pm 1$, all of which are clearly algebraic, and the number of which is finite. Denote all such t values in (t_1, t_2) by $\tau_1, \tau_2, \ldots, \tau_l$. We also find all zeros of

$$F'(t) = 1 - \frac{\alpha'(t)}{\sqrt{1 - \alpha(t)^2}} + \frac{\delta'(t)}{\sqrt{1 - \delta(t)^2}} = 0$$

in (t_1, t_2), and denote them $\sigma_1, \sigma_2, \ldots, \sigma_m$. They are also algebraic and finite in number. Note that $\beta(t) = 0$ implies $b(t) = 0$, and hence $\alpha(t) = \pm 1$.

Now F and β have the following properties:

(1) F and β are continuous on $[t_1, t_2]$.
(2) F is strictly monotone on each subinterval generated by $\tau_1, \ldots, \tau_l$, $\sigma_1, \ldots, \sigma_m$, and the function β does not change sign on the subintervals.

So the decision problem on the existence of a zero of F in $[t_1, t_2]$ can be resolved by determining the signs of F at the extremal points $t_1, t_2, \tau_1, \ldots, \tau_l, \sigma_1, \ldots, \sigma_m$ and by checking (1) either one of them is a zero or endpoints of the subintervals have opposite signs, and (2) the sign of β at the zero or the signs of endpoints of the subinterval that contains a zero of F.

The sign determination of β on the algebraic points is clearly decidable, since β is algebraic. The sign determination for F can be done exactly as well, since we can determine the zero problem for the following expression with an algebraic t_* [2, 8]:

$$F(t_*) = t_* + \arccos \alpha(t_*) - \arccos \delta(t_*) - 2k \arccos(-1). \tag{10}$$

Clearly, similar procedures work for the other equations (6)–(8).

3 Complexity

In this section, we calculate an explicit bit complexity for our problem. Although the decidability result in Section 2 is valid for any *algebraic* input trajectory $c(t)$, we assume in this section that $c(t)$ is given by polynomials with rational coefficients.

Assumption: The functions $c_1(t)$, $c_2(t)$, $c_3(t)$, which define the trajectory of the moving ball $\mathcal{B}$'s center, are in $\mathbb{Q}[t]$. Also, the constant s and the ball's radius r are rational numbers.

See [1, 9] for more details on the notions introduced in this section.

3.1 Input Size

Let $f(t) = a_n t^n + a_{n-1} t^{n-1} + \cdots + a_0 \in \mathbb{C}[t]$ with $a_n \neq 0$. The *Mahler measure* of f, $M(f)$ is defined by

$$M(f) := |a_n| \cdot \prod_{i=1}^{n} \max\{1, |\gamma_i|\},$$

where $\gamma_1, \ldots, \gamma_n \in \mathbb{C}$ are the zeros of f. It follows from the definition that the Mahler measure is a multiplicative map from $\mathbb{C}[t]$ to $\{x \in \mathbb{R} \mid x > 0\}$, *i.e.*,

$$M(f_1 f_2) = M(f_1) M(f_2), \quad \forall f_1, f_2 \in \mathbb{C}[t]. \tag{11}$$

The *absolute logarithmic height* of f, $\mathrm{ht}(f)$, is defined by

$$\mathrm{ht}(f) := \frac{1}{\deg(f)} \log M(f).$$

Let γ be an algebraic number, and let $f \in \mathbb{Z}[t]$ be its minimal polynomial. The *degree* $\deg(\gamma)$, the *Mahler measure* $M(\gamma)$, and the *absolute logarithmic height* $\mathrm{ht}(\gamma)$ are defined respectively by:

$$\deg(\gamma) := \deg(f), \quad M(\gamma) := M(f), \quad \mathrm{ht}(\gamma) := \mathrm{ht}(f).$$

Here are some properties of the absolute logarithmic height:

Lemma 5. *Let $\gamma, \gamma_1, \ldots, \gamma_n$ be (nonzero) algebraic numbers. Then we have*

(1) $\mathrm{ht}(\gamma_1\gamma_2) \le \mathrm{ht}(\gamma_1) + \mathrm{ht}(\gamma_2)$.
(2) $\mathrm{ht}(\gamma_1 + \cdots + \gamma_n) \le \mathrm{ht}(\gamma_1) + \cdots + \mathrm{ht}(\gamma_n) + \log n$.
(3) $\mathrm{ht}(\gamma^r) = |r| \cdot \mathrm{ht}(\gamma)$, $\forall r \in \mathbb{Q}$.

Proof. See [9]. □

We use the degrees of the input polynomials as a measure of the input size:

Input Condition 1: $\deg(c_1), \deg(c_2), \deg(c_3) \le N$.

For the second measure of input size, consider the following: Let

$$f(t) = \frac{p_n}{q_n} t^n + \frac{p_{n-1}}{q_{n-1}} t^{n-1} + \cdots + \frac{p_0}{q_0} \in \mathbb{Q}[t],$$

where $p_n, q_0, \ldots, q_n \neq 0$, and $(p_i, q_i) = 1$ for $i = 0, 1, \ldots, n$. We define the *bit bound* of f, $B(f)$, by

$$B(f) := \max_{0 \le i \le n} \{\log_2 |p_i|, \log_2 |q_i|\}.$$

Input Condition 2: $B(c_1), B(c_2), B(c_3) \le B$ and $B(s), B(r) \le B$.

The final bit complexity will be expressed in terms of these two input parameters N and B. Here are some properties of the bit bound:

Lemma 6. *For any $f(t), g(t) \in \mathbb{Q}[t]$, we have*

(1) $B(f \pm g) \le B(f) + B(g) + 1$.
(2) $B(fg) \le (N+1)\log_2(N+1) \cdot (B(f) + B(g))$, *where* $N = \min\{\deg(f), \deg(g)\}$.

(3) $B(f') \leq B(f)\log_2(\deg(f))$.
(4) $M(f) \leq \sqrt{1+\deg(f)} \cdot 2^{B(f)}$.

Proof. See [9] for the proof of (4). Let $n = \deg(f)$, $m = \deg(g)$, $B_f = B(f)$, $B_g = B(g)$. Write

$$f(t) = \frac{p_n}{q_n}t^n + \frac{p_{n-1}}{q_{n-1}}t^{n-1} + \cdots + \frac{p_0}{q_0}, \quad g(t) = \frac{\tilde{p}_m}{\tilde{q}_m}t^n + \frac{\tilde{p}_{n-1}}{\tilde{q}_{n-1}}t^{n-1} + \cdots + \frac{\tilde{p}_0}{\tilde{q}_0},$$

where $|p_i|, |q_i| \leq 2^{B_f}$, $|\tilde{p}_i|, |\tilde{q}_i| \leq 2^{B_g}$. Note that a coefficient of $f(t) \pm g(t)$ is of the form:

$$\frac{p}{q} \pm \frac{\tilde{p}}{\tilde{q}} = \frac{p\tilde{q} \pm \tilde{p}q}{q\tilde{q}}.$$

So (1) follows, since $|p\tilde{q} \pm \tilde{p}q|, |q\tilde{q}| \leq 2 \cdot 2^{B_f+B_g} = 2^{B_f+B_g+1}$.

Let $N = \min\{n, m\}$. Note that a coefficient of $f(t)g(t)$ is of the form:

$$\sum_{k=0}^{L} \frac{p_{i_k}}{q_{i_k}} \cdot \frac{\tilde{p}_{j_k}}{\tilde{q}_{j_k}} = \frac{p_{i_0}q_{i_1}\cdots q_{i_L} \cdot \tilde{p}_{i_0}\tilde{q}_{i_1}\cdots\tilde{q}_{i_L} + \cdots + q_{i_0}\cdots q_{i_{L-1}}p_{i_L} \cdot \tilde{q}_{i_0}\cdots\tilde{q}_{i_{L-1}}\tilde{p}_{i_L}}{\prod_{k=0}^{L} q_{i_k}\tilde{q}_{j_k}},$$

for some $L < N$. The bit size of the above number is less than $\log_2\left\{(N+1)2^{(N+1)B_f}2^{(N+1)B_g}\right\}$, from which (2) follows.

Finally, note that the coefficients of $f'(t)$ are:

$$i \cdot \frac{p_i}{q_i}, \quad 0 \leq i \leq n.$$

So (3) follows, since $|i \cdot p_i|, |q_i| \leq n \cdot 2^{B_f}$. □

3.2 Results From Transcendental Number Theory

The following Baker type result is an effective tool for the zero problem of transcendental expressions like (10):

Proposition 7 (Waldschmidt [8], Theorem C). *For $n \geq 2$, let $\gamma_0, \gamma_1, \cdots, \gamma_n$ be algebraic numbers, and let $\beta_1, \cdots, \beta_n$ be nonzero algebraic numbers. For $1 \leq j \leq n$, let $\log\beta_j$ be any determination of the logarithm of β_j. Suppose that*

$$\begin{gathered} D \geq [\mathbb{Q}(\gamma_0, \gamma_1, \cdots, \gamma_n, \beta_1, \cdots, \beta_n) : \mathbb{Q}],\ W \geq \max_{0\leq j\leq n}\{\mathrm{ht}(\gamma_j)\}, \\ V_j \geq \max\{\mathrm{ht}(\beta_j), |\log\beta_j|/D, 1/D\},\ V_1 \leq \cdots \leq V_n, \\ V_{n-1}^+ = \max\{V_{n-1}, 1\},\ V_n^+ = \max\{V_n, 1\}. \\ 1 < E \leq \min\{e^{DV_1}, \min_{1\leq j\leq n}\{4DV_j/|\log\beta_j|\}\}. \end{gathered}$$

If $\Lambda := \gamma_0 + \gamma_1\log\beta_1 + \cdots + \gamma_n\log\beta_n$ is non-zero, then

$$|\Lambda| > \exp\{-2^{8n+51}n^{2n}D^{n+2}V_1\cdots V_n(W + \log(EDV_n^+))(\log(EDV_{n-1}^+))(\log E)^{-n-1}\}.$$

By applying Proposition 7 after replacing $\gamma_0 \to i\gamma_0$, $\beta_j \to \beta_j + i\sqrt{1-\beta_j^2}$, $1 \le j \le n$, we transform Proposition 7 into the following form, which is suitable to our situation:

Corollary 8. *Let $\gamma_0, \gamma_1, \ldots, \gamma_n, \beta_1, \ldots, \beta_n \in \mathbb{C}$ ($n \ge 2$) be nonzero algebraic numbers. If $\Lambda := \gamma_0 + \gamma_1 \arccos\beta_1 + \cdots + \gamma_n \arccos\beta_n$ is non-zero, then*

$$|\Lambda| > \exp\{-2^{8n+51}n^{2n}D^{n+2}V_1\cdots V_n(W+\log(EDV_n^+))(\log(EDV_{n-1}^+))(\log E)^{-n-1}\},$$

where

$$D \ge 2^{n+1}\deg(\gamma_0)\cdots\deg(\gamma_n)\cdot\deg(\beta_1)\cdots\deg(\beta_n),\ W \ge \max_{0\le j\le n}\{\mathrm{ht}(\gamma_j)\},$$
$$V_j \ge \max\{2\mathrm{ht}(\beta_j) + \frac{3}{2}\log 2, |\arccos\beta_j|/D, 1/D\},\ V_1 \le \cdots \le V_n,$$
$$V_{n-1}^+ = \max\{V_{n-1}, 1\},\ V_n^+ = \max\{V_n, 1\},$$
$$1 < E \le \min\{e^{DV_1}, \min_{1\le j\le n}\{4DV_j/|\arccos\beta_j|\}\}.$$

3.3 Asymptotic Bit Complexity

Now we bound the bit complexity of the expression from (10)

$$\Lambda := F(t_*) = t_* + \arccos\alpha(t_*) - \arccos(\delta(t_*)) - 2k\arccos(-1), \tag{12}$$

where t_* is a zero of one of the following (algebraic) functions: (i) $\alpha(t) \pm 1$, (ii) $\delta(t) \pm 1$, (iii) $F'(t)$. The complexity argument for the other cases arising from (6), (7), (8) would be identical. To apply Corollary 8, we first need to bound the following quantities:

$$\deg(t_*),\quad \deg(\alpha(t_*)),\quad \deg(\delta(t_*)),\quad \mathrm{ht}(t_*),\quad \mathrm{ht}(\alpha(t_*)),\quad \mathrm{ht}(\delta(t_*)),\quad \mathrm{ht}(2k). \tag{13}$$

Lemma 9. *Let γ be a zero of $f[t] \in \mathbb{Q}[t]$. Then we have:*

(1) $\deg(\gamma) \le \deg(f)$.
(2) $\mathrm{ht}(\gamma) \le (\deg(f)+2)B(f)\log 2 + \frac{1}{2}\log(\deg(f)+1)$.

Proof. (1) is obvious. For the proof of (2), let $f_\gamma(t) \in \mathbb{Z}[t]$ be the minimal polynomial of γ. Let $n = \deg(f)$ and $B = B(f)$. Write

$$f(t) = \frac{p_n}{q_n}t^n + \frac{p_{n-1}}{q_{n-1}}t^{n-1} + \cdots + \frac{p_0}{q_0},$$

where $p_n, q_0, \ldots, q_n \ne 0$, and $\max_{i=0}^n\{|p_i|, |q_i|\} \le 2^B$. Note that

$$\begin{aligned} f(t) &= \frac{1}{q_0q_1\cdots q_n}\cdot\left(q_0\cdots q_{n-1}p_nt^n + q_0\cdots q_{n-2}p_{n-1}q_nt^{n-1} + \cdots + p_0q_1\cdots q_n\right)\\ &= \frac{1}{q_0q_1\cdots q_n}\cdot f_\gamma(t)g(t), \end{aligned}$$

for some $g(t) \in \mathbb{Z}[t]$. From (11), we have $M(q_0\cdots q_n)\cdot M(f) = M(f_\gamma)M(g)$, and hence,

$$\begin{aligned} M(f_\gamma) &\le M(q_0)\cdots M(q_n)\cdot M(f) = |q_0|\cdots|q_n|\cdot M(f) \le 2^{(n+1)B}\cdot M(f)\\ &\le 2^{(n+1)B}\cdot\sqrt{1+n}\cdot 2^B = 2^{(n+2)B}\sqrt{1+n}, \end{aligned}$$

where the last inequality comes from Lemma 6 (4). Now we have

$$\begin{aligned}\mathrm{ht}(\gamma) &= \frac{1}{\deg(f_\gamma)}\log(M(f_\gamma)) \le \log(M(f_\gamma))\\ &\le \log\left(2^{(n+2)B}\sqrt{1+n}\right) = (n+2)B\log 2 + \frac{1}{2}\log(n+1),\end{aligned}$$

which completes the proof. □

Lemma 10. *Let γ be an algebraic number, and let $g(t) \in \mathbb{Q}[t]$. Suppose that* $\mathrm{ht}(\gamma) \ge 1$. *Then we have*

$$\mathrm{ht}(g(\gamma)) \le \frac{1}{2}\deg(g)\left(\deg(g)+1\right)\cdot \mathrm{ht}(\gamma) + 2\log 2\cdot(\deg(g)+1)B(g) + \log(\deg(g)+1). \quad (14)$$

Proof. Let $n = \deg(g)$ and $B = B(g)$. Write

$$g(t) = \frac{p_n}{q_n}t^n + \frac{p_{n-1}}{q_{n-1}}t^{n-1} + \cdots + \frac{p_0}{q_0},$$

where $p_n, q_0, \ldots, q_n \neq 0$ and $\max_{i=0}^n\{|p_i|, |q_i|\} \le 2^B$. Now we have

$$\begin{aligned}\mathrm{ht}(g(\gamma)) &= \mathrm{ht}\left(\frac{p_n}{q_n}\gamma^n + \frac{p_{n-1}}{q_{n-1}}\gamma^{n-1} + \cdots + \frac{p_0}{q_0}\right)\\ &\le \mathrm{ht}\left(\frac{p_n}{q_n}\cdot\gamma^n\right) + \mathrm{ht}\left(\frac{p_{n-1}}{q_{n-1}}\cdot\gamma^{n-1}\right) + \cdots + \mathrm{ht}\left(\frac{p_0}{q_0}\right) + \log(n+1)\\ &\le \sum_{i=1}^n \mathrm{ht}\left(\gamma^i\right) + \sum_{i=0}^n \left(\mathrm{ht}\left(p_i\right) + \mathrm{ht}\left(q_i\right)\right\} + \log(n+1)\\ &\le \mathrm{ht}\left(\gamma\right)\cdot\sum_{i=1}^n i + \sum_{i=0}^n 2\log\left(2^B\right) + \log(n+1)\\ &\le \frac{1}{2}n(n+1)\cdot\mathrm{ht}(\gamma) + 2\log 2\cdot(n+1)B + \log(n+1),\end{aligned}$$

which completes the proof. □

Lemma 11. *The following hold for the polynomials $a(t)$, $b(t)$, and $d(t)$:*

$$\begin{gathered}B(a) = O(B), \qquad B(b) = O(B), \qquad B(d) = O(BN\log N),\\ B(a') = O(B\log N), B(b') = O(B\log N), B(d') = O\left(BN(\log N)^2\right).\end{gathered}$$

Proof. Use the relations between a, b, d and c_1, c_2, c_3, and Lemma 6, along with the following:

$$\begin{aligned}B(d) &\le B\left(c_1^2 + c_2^2 + (st-c_3)^2 + 1 - r^2\right)\\ &\le B\left(c_1^2\right) + B\left(c_2^2\right) + B\left((st-c_3)^2\right) + B\left(r^2\right) + 4\\ &\le 2\cdot(N+1)\log_2(N+1)\cdot 2B + (N+1)\log_2(N+1)\cdot 2B(st-c_3) + 2B + 4\\ &= O(BN\log N),\\ B(d') &\le B\left(2c_1c_1' + 2c_2c_2' + 2\left(st-c_3\right)\left(s-c_3'\right)\right)\\ &\le 1 + B\left(c_1c_1' + c_2c_2' + \left(st-c_3\right)\left(s-c_3'\right)\right)\\ &\le B\left(c_1c_1'\right) + B\left(c_2c_2'\right) + B\left(\left(st-c_3\right)\left(s-c_3'\right)\right) + 3\\ &\le 2\cdot N\log_2 N\cdot\left(B + O\left(B\log N\right)\right) + N\log_2 N\cdot\left(B + O\left(B\log N\right)\right) + 3\\ &= O\left(BN\left(\log N\right)^2\right).\end{aligned}$$

$\square$

By using Lemmas 5, 6, 9, 10, and 11, we obtain the following estimates for the quantities in (13) for each of the cases (i) $\alpha(t_*) \pm 1 = 0$, (ii) $\delta(t_*) \pm 1 = 0$, and (iii) $F'(t_*) = 0$.

(i) When $\alpha(t_*) \pm 1 = 0$**:**

$$\alpha(t_*) = \frac{a(t_*)}{\sqrt{a(t_*)^2 + b(t_*)^2}} = \pm 1 \quad \to \quad a(t_*)^2 = a(t_*)^2 + b(t_*)^2 \quad \to \quad b(t_*) = 0 \quad \to \quad c_2(t_*) = 0.$$

Note that $\deg(c_2) \le N$ and $B(c_2) \le B$. So we have:

$$\deg(t_*) \le \deg(c_2) \le N, \tag{15}$$
$$\deg(\alpha(t_*)) = \deg(\pm 1) = 1, \tag{16}$$
$$\deg(\delta(t_*)) = \deg\left(\frac{d(t_*)}{a(t_*)}\right) \le \deg(t_*) \le N, \tag{17}$$

$$\begin{aligned}
\mathrm{ht}(t_*) &\le (\deg(c_2) + 2)B(c_2)\log 2 + \frac{1}{2}\log(\deg(c_2) + 1) \le (N+2)B\log 2 + \frac{1}{2}\log(N+1) \\
&= O(BN), && (18) \\
\mathrm{ht}(\alpha(t_*)) &= \mathrm{ht}(\pm 1) = 0, && (19) \\
\mathrm{ht}(\delta(t_*)) &= \mathrm{ht}\left(\frac{d(t_*)}{\sqrt{a(t_*)^2 + b(t_*)^2}}\right) = \mathrm{ht}\left(\frac{d(t_*)}{a(t_*)}\right) \le \mathrm{ht}(a(t_*)) + \mathrm{ht}(d(t_*)) \\
&\le \frac{1}{2}\deg(a)\left(\deg(a)+1\right)\cdot \mathrm{ht}(t_*) + 2\log 2 \cdot \left(\deg(a)+1\right)B(a) + \log\left(\deg(a)+1\right) \\
&\quad + \frac{1}{2}\deg(d)\left(\deg(d)+1\right)\cdot \mathrm{ht}(t_*) + 2\log 2 \cdot \left(\deg(d)+1\right)B(d) + \log\left(\deg(d)+1\right) \\
&\le \left\{\frac{1}{2}N(N+1) + N(2N+1)\right\}\cdot O\left(BN\right) + 2\log 2\cdot (N+1)\cdot O(B) \\
&\quad + 2\log 2 \cdot (2N+1)\cdot O(BN\log N) + \log\left(N+1\right) + \log\left(2N+1\right) \\
&= O\left(BN^3\right). && (20)
\end{aligned}$$

(ii) When $\delta(t_*) \pm 1 = 0$**:**

$$\to \quad \delta(t_*) = \frac{d(t_*)}{\sqrt{a(t_*)^2 + b(t_*)^2}} = \pm 1 \quad \to \quad a(t_*)^2 + b(t_*)^2 - d(t_*)^2 = 0$$

Let $u(t) := a(t)^2 + b(t)^2 - d(t)^2 \in \mathbb{Q}[t]$. Note that $\deg(u) \le 4N$ and

$$\begin{aligned}
B(u) &= B\left(a^2 + b^2 - d^2\right) \le B\left(a^2\right) + B\left(b^2\right) + B\left(d^2\right) + 2 \\
&\le 2\cdot (N+1)\log_2(N+1)\cdot 2B + (2N+1)\log_2(2N+1)\cdot 2B(d) + 2 \\
&= O\left(BN^2\left(\log N\right)^2\right)
\end{aligned}$$

So we have

$$\deg(t_*) \le \deg\left(u\right) \le 4N, \tag{21}$$
$$\deg(\alpha(t_*)) = \deg\left(\frac{a(t_*)}{\sqrt{a(t_*)^2 + b(t_*)^2}}\right) \le \deg(t_*) \le 4N, \tag{22}$$
$$\deg(\delta(t_*)) = \deg(\pm 1) = 1, \tag{23}$$

$$\begin{aligned}
\mathrm{ht}(t_*) &\le (\deg(u)+2)B(u)\cdot\log 2 + \frac{1}{2}\log(\deg(u)+1)\\
&\le (4N+2)\log 2\cdot O\left(BN^2(\log N)^2\right) + \frac{1}{2}\log(4N+1) = O\left(BN^3(\log N)^2\right), \qquad (24)\\
\mathrm{ht}(\alpha(t_*)) &= \mathrm{ht}\left(\frac{a(t_*)}{\sqrt{a(t_*)^2+b(t_*)^2}}\right) = \mathrm{ht}\left(\frac{a(t_*)}{d(t_*)}\right) \le \mathrm{ht}(a(t_*)) + \mathrm{ht}(d(t_*))\\
&\le \frac{1}{2}\deg(a)\left(\deg(a)+1\right)\cdot\mathrm{ht}(t_*) + 2\log 2\cdot\left(\deg(a)+1\right)B(a) + \log\left(\deg(a)+1\right)\\
&\quad + \frac{1}{2}\deg(d)\left(\deg(d)+1\right)\cdot\mathrm{ht}(t_*) + 2\log 2\cdot\left(\deg(d)+1\right)B(d) + \log\left(\deg(d)+1\right)\\
&\le \left\{\frac{1}{2}N(N+1) + N(2N+1)\right\}\cdot O\left(BN^3(\log N)^2\right) + 2\log 2\cdot(N+1)\cdot O(B)\\
&\quad + 2\log 2\cdot(2N+1)\cdot O(BN\log N) + \log(N+1) + \log(2N+1)\\
&= O\left(BN^5(\log N)^2\right), \qquad (25)\\
\mathrm{ht}(\delta(t_*)) &= \mathrm{ht}(\pm 1) = 0. \qquad (26)
\end{aligned}$$

(iii) When $F'(t_*) = 0$: Note that

$$\begin{aligned}
F'(t) &= 1 - \frac{\alpha(t)}{\sqrt{1-\alpha(t)^2}} + \frac{\delta(t)}{\sqrt{1-\delta(t)^2}}\\
&= 1 - \frac{\frac{a'(t)\sqrt{a(t)^2+b(t)^2} - a(t)\frac{a(t)a'(t)+b(t)b'(t)}{\sqrt{a(t)^2+b(t)^2}}}{a(t)^2+b(t)^2}}{\sqrt{1-\frac{a(t)^2}{a(t)^2+b(t)^2}}} + \frac{\frac{d'(t)\sqrt{a(t)^2+b(t)^2} - d(t)\frac{a(t)a'(t)+b(t)b'(t)}{\sqrt{a(t)^2+b(t)^2}}}{a(t)^2+b(t)^2}}{\sqrt{1-\frac{d(t)^2}{a(t)^2+b(t)^2}}}\\
&= \frac{1}{\left\{a(t)^2+b(t)^2\right\}\sqrt{a(t)^2+b(t)^2-d(t)^2}}\\
&\quad\cdot\Big[\left\{\left(a(t)^2+b(t)^2\right) - \left(a'(t)b(t)-a(t)b'(t)\right)\right\}\sqrt{a(t)^2+b(t)^2-d(t)^2}\\
&\qquad + \left\{d'(t)\left(a(t)^2+b(t)^2\right) - d(t)\left(a'(t)a(t)+b'(t)b(t)\right)\right\}\Big]
\end{aligned}$$

So we have $v(t_*) = 0$, where

$$\begin{aligned}
v(t) :=& \left\{a(t)^2+b(t)^2-a'(t)b(t)+a(t)b'(t)\right\}^2\left\{a(t)^2+b(t)^2-d(t)^2\right\}\\
&- \left\{d'(t)\left(a(t)^2+b(t)^2\right) - d(t)\left(a'(t)a(t)+b'(t)b(t)\right)\right\}^2 \qquad \in \mathbb{Q}[t].
\end{aligned}$$

Note that $\deg(v) \le 8N$, and

$$\begin{aligned}
B(v) &\le (4N+1)\log_2(4N+1)\cdot\left\{B\left(\left(a^2+b^2-a'b+ab'\right)^2\right)+B(u)\right\}\\
&\quad +4N\log_2(4N)\cdot 2B\left(d'(a^2+b^2)-d(aa'+bb')\right)+1\\
&\le (4N+1)\log_2(4N+1)\cdot\left\{(2N+1)\log_2(2N+1)\cdot 2B\left(a^2+b^2-a'b+ab'\right)\right.\\
&\qquad \left.+O\left(BN^2(\log N)^2\right)\right\}\\
&\quad +4N\log_2(4N)\cdot 2\left\{B\left(d'(a^2+b^2)\right)+B\left(d(aa'+bb')\right)+1\right\}+1\\
&\le O\left(N^2(\log N)^2\right)\cdot\left\{B\left(a^2\right)+B\left(b^2\right)+B\left(a'b\right)+B\left(ab'\right)+3\right\}+O\left(BN^3(\log N)^3\right)\\
&\quad +O\left(N\log N\right)\cdot 2N\log_2(2N)\cdot\left\{B\left(d'\right)+B\left(a^2\right)+B\left(b^2\right)+B(d)+B\left(aa'\right)+B\left(bb'\right)+2\right\}+1\\
&\le O\left(N^2(\log N)^2\right)\cdot O\left(BN(\log N)^2\right)+O\left(BN^3(\log N)^3\right)\\
&\quad +O\left(N\log N\right)\cdot 2N\log_2(2N)\cdot O\left(BN(\log N)^2\right)+1\\
&= O\left(BN^3(\log N)^4\right).
\end{aligned}$$

So we have

$$\deg(t_*) \le \deg(v) \le 8N, \tag{27}$$

$$\deg(\alpha(t_*)) = \deg\left(\frac{a(t_*)}{\sqrt{a(t_*)^2+b(t_*)^2}}\right) \le \deg(t_*) \le 8N, \tag{28}$$

$$\deg(\delta(t_*)) = \deg\left(\frac{d(t_*)}{\sqrt{a(t_*)^2+b(t_*)^2}}\right) \le \deg(t_*) \le 8N, \tag{29}$$

$$\mathrm{ht}(t_*) \le (\deg(v)+2)B(v)\log 2+\frac{1}{2}\log(\deg(v)+1) = O\left(BN^4(\log N)^4\right), \tag{30}$$

$$\begin{aligned}
\mathrm{ht}(\alpha(t_*)) &= \mathrm{ht}\left(\frac{a(t_*)}{\sqrt{a(t_*)^2+b(t_*)^2}}\right) \le \mathrm{ht}\left(a(t_*)\right)+\mathrm{ht}\left(\sqrt{a(t_*)^2+b(t_*)^2}\right)\\
&= \mathrm{ht}(a(t_*))+\frac{1}{2}\mathrm{ht}\left(a(t_*)^2+b(t_*)^2\right) \le \mathrm{ht}(a(t_*))+\frac{1}{2}\left\{\mathrm{ht}\left(a(t_*)^2\right)+\mathrm{ht}\left(b(t_*)^2\right)+\log 2\right\}\\
&= 2\mathrm{ht}(a(t_*))+\mathrm{ht}(b(t_*))+\frac{1}{2}\log 2\\
&\le 3\cdot\left\{\frac{1}{2}N(N+1)\cdot\mathrm{ht}(t_*)+2\log 2(N+1)B+\log(N+1)\right\}+\frac{1}{2}\log 2\\
&= O\left(BN^6(\log N)^4\right),
\end{aligned} \tag{31}$$

$$\begin{aligned}
\mathrm{ht}(\delta(t_*)) &= \mathrm{ht}\left(\frac{d(t_*)}{\sqrt{a(t_*)^2+b(t_*)^2}}\right) \le \mathrm{ht}(d(t_*))+\mathrm{ht}(a(t_*))+\mathrm{ht}(b(t_*))+\frac{1}{2}\log 2\\
&= O\left(\mathrm{ht}(d(t_*))\right) \qquad (32)\\
&= O\left(\frac{1}{2}\cdot 2N(2N+1)\cdot\mathrm{ht}(t_*)+2\log 2(2N+1)\cdot O\left(BN\log N\right)+\log(2N+1)\right)\\
&= O\left(BN^6(\log N)^4\right). \qquad (33)
\end{aligned}$$

Lemma 12. *The constant k in (12) is bounded as:* $k = O\left(2^{BN^2(\log N)^2}\right)$.

Proof. t_1 and t_2 in

$$t_1 - 2\pi \le t + \arccos\alpha(t) - \arccos(-\delta(t)) \le t_2 + 2\pi$$

are zeros of $u(t) = a(t)^2 + b(t)^2 - d(t)^2$. So by using the Cauchy bound, we have

$$|2k| \le \frac{|t_i|}{\pi} + 2 \le \frac{1 + 2^{2B(u)}}{\pi} + 2 = O\left(2^{BN^2(\log N)^2}\right).$$

□

Now we are ready for the following bit complexity bound:

Theorem 13. *The sign of Λ in (12) can be determined using $O\left(B^3 \log B \cdot N^{28} (\log N)^{13}\right)$ bits.*

Proof. Note that $\Lambda = t_* + \arccos\alpha(t_*) - \arccos(\delta(t_*)) - 2k\arccos(-1)$. To apply Corollary 8, we let $n = 3$, $\gamma_0 = t_*$, $\gamma_1 = 1$, $\gamma_2 = -1$, $\gamma_3 = -2k$, $\beta_1 = \alpha(t_*)$, $\beta_2 = \delta(t_*)$, $\beta_3 = -1$. From (15)–(33), we have

$$\deg(\gamma_0) = O(N), \quad \deg(\gamma_1) = \deg(\gamma_2) = \deg(\gamma_3) = 1,$$

$$\deg(\beta_1) = \deg(\beta_2) = O(N), \quad \deg(\beta_3) = 1,$$

$$\mathrm{ht}(\gamma_0) = O\left(BN^4 (\log N)^4\right), \quad \mathrm{ht}(\gamma_1) = \mathrm{ht}(\gamma_2) = 0, \quad \mathrm{ht}(\gamma_3) = O\left(BN^2 (\log N)^2\right),$$

$$\mathrm{ht}(\beta_1) = \mathrm{ht}(\beta_2) = O\left(BN^6 (\log N)^4\right), \quad \mathrm{ht}(\beta_3) = 0.$$

Since

$$2^4 \deg(\gamma_0)\deg(\gamma_1)\deg(\gamma_2)\deg(\gamma_3)\deg(\beta_1)\deg(\beta_2)\deg(\beta_3) = O\left(N^3\right),$$
$$\max\{\mathrm{ht}(\gamma_0), \mathrm{ht}(\gamma_1), \mathrm{ht}(\gamma_2), \mathrm{ht}(\gamma_3)\} = O\left(BN^4 (\log N)^4\right),$$

we take $D = C_1 \cdot N^3$ and $W = C_2 \cdot BN^4 (\log N)^4$ for some positive constants C_1, C_2. Note that $\max\left\{2\mathrm{ht}(\beta_i) + \frac{3}{2}\log 2, \arccos\beta_i/D, 1/D\right\} \le \max\left\{O\left(BN^6 (\log N)^4\right), \pi/\left(C_1 N^3\right)\right\} = O\left(BN^6 (\log N)^4\right)$ for $i = 1, 2$, and $\max\left\{2\mathrm{ht}(\beta_3), \arccos\beta_3/D, 1/D\right\} \le \max\left\{0, \pi/\left(C_1 N^3\right)\right\} \le C_3 \cdot N^{-3}$ for some positive constant C_3. So we take $V_1 = C_3 \cdot N^{-3}$ and $V_2 = V_2^+ = V_3 = V_3^+ = C_4 BN^6 (\log N)^4$. Since $\min\left\{e^{DV_1}, \min_{1\le j\le 3}\left\{4DV_j/\left|\arccos\beta_j\right|\right\}\right\} \ge \min\left\{e^{C_1C_3}, (1/\pi)\cdot\min\left\{4C_1C_3, 4C_1C_4 BN^9 (\log N)^4\right\}\right\}$, we can take $E = C_5$ for some constant $C_5 > 1$. Now by Corollary 8, we have

$$\begin{aligned}
&-\log(|\Lambda|)\\
&< C \cdot D^5 V_1 V_2 V_3 \cdot \left\{W + \log\left(EDV_3^+\right)\right\} \cdot \log\left(EDV_2^+\right) \cdot (\log E)^{-4}\\
&= C \cdot \left(C_1 N^3\right)^5 \cdot \left\{C_3 C_4^2 B^2 N^9 (\log N)^8\right\} \cdot \left\{C_2 BN^4 (\log N)^4 + \log\left(C_1C_4C_5 BN^9 (\log N)^4\right)\right\}\\
&\qquad \cdot \log\left(C_1C_4C_5 BN^9 (\log N)^4\right) \cdot (\log C_5)^{-4}\\
&= O\left(N^{15}\right) \cdot O\left(B^2 N^9 (\log N)^8\right) \cdot O\left(BN^4 (\log N)^4\right) \cdot O(\log B + \log N)\\
&= O\left(B^3 N^{28} (\log N)^{12} \cdot (\log B + \log N)\right) = O\left(B^3 \log B \cdot N^{28} (\log N)^{13}\right)
\end{aligned}$$

□

Remark 14. The number of times we need to determine the signs of such Λ's is bounded by:

$$\left(\# \text{ zeros of } \alpha \pm 1, \delta \pm 1, F'\right) \leq (\# \text{ zeros of } b, u, w) = O(N).$$

Remark 15. We have no intention to claim the asymptotic bound in Theorem 13 is the best we can get: The various estimation in Section 3 are rather 'generous'. Furthermore, there has been some improvements [4] for Waldschmidt's result, Proposition 7.

4 More General Cases

The method in Section 2 covers slightly more general situations. For example, the motion of the point p can be of the form: $h(t) = (\cos t, \sin t, st) + (p_1(t), p_2(t), p_3(t))$, where $p_i(t)$'s are algebraic functions. Because non-circular part of the motion can be absorbed by $c(t)$, we obtain the same form of equation as (2). Also considering two balls, instead of a ball and a point, in the same type of motions does not change the form of the equation: say the radii of the balls are r_1 and r_2, then we want to check the equation $\|h(t) - c(t)\| \leq r_1 + r_2$.

However, a fundamentally different situation can occur if we consider, for example, an elliptic motion instead of a circular motion. The second-degree trigonometric functions do not cancel out as in (2). So the corresponding equation would look like

$$\rho \cos 2t + a(t) \cos t + b(t) \sin t + d(t) = 0, \tag{34}$$

for a constant ρ. This equation is not reduced to the form of (3) to which we can apply the zero bound discussed in Section 3.2. Currently, we do not know how to deal with equations of this type.

A similar difficulty arises if we consider a more general semi-algebraic body. Even if we consider a pure helical motion of a point and an algebraically parametrized motion, the collision equation that we have to deal with may involve higher-degree trigonometric functions, for example, upto $\sin dt$, where d is the degree of a polynomial (among possibly many) that defines the semi-algebraic body.

References

1. A. Baker. *Transcendental Number Theory.* Cambridge University Press, 1979.
2. E.-C. Chang, S. W. Choi, D. Kwon, H. Park, and C. K. Yap. Shortest paths for disc obstacles is computable. In *21st ACM Symp. on Comp. Geometry*, pages 116–125, 2005. June 5-8, Pisa, Italy.
3. M. Hofer, B. Odehnal, H. Pottmann, T. Steiner, and J. Wallner. 3D shape recognition and reconstruction based on line element geometry. In *10th IEEE Intl. Conf. on Computer Vision (ICCV'05)*, volume 2, pages 1532–1538, 2005.
4. E. M. Matveev. An explicit lower bound for a homogeneous rational linear form in logarithms of algebraic numbers. *Izvestiya, Mathematics*, 62(4):723–772, 1998.
5. S. Mick and O. Röschel. Interpolation of helical patches by kinematic rational Bézier patches. *Computers and Graphics*, 14(2):275–280, 1990.
6. J. T. Schwartz and M. Sharir. On the piano movers' problem: II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Appl. Math.*, 4:298–351, 1983.
7. V. Shapiro. Solid modeling. In G. Farin, J. Hoschek, and M.-S. Kim, editors, *Handbook of Computer Aided Geometric Design*. North-Holland, Amsterdam, 2002.
8. M. Waldschmidt. Transcendence measures for exponentials and logarithms. *J. Austral. Math. Soc. Ser. A*, 25(4):445–465, 1978.
9. M. Waldschmidt. *Diophantine Approximation on Linear Algebraic Groups.* Series of Comprehensive Studies in Mathematics, Vol. 328. Springer, Berlin, 2000.
10. C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004. Expanded from 1997 version.

Software Techniques for Perfect Elementary Functions in Floating-Point Interval Arithmetic

Florent de Dinechin[1] and Sergey Maidanov[2]

[1] École Normale Supérieure de Lyon
Florent.de.Dinechin@ens-lyon.fr
[2] Intel Corporation
Sergey.Maidanov@intel.com

Abstract. A few recent processors allow very efficient double-precision floating point interval arithmetic for the basic operations. In comparison, the elementary functions available in current interval arithmetic packages suffer two major weaknesses. The first is accuracy, as intervals grow more in an elementary function than what is mathematically possible considering only the number format. The second is performance, well below that of scalar elementary function libraries. The difficulty is that the need to prove the containment property for interval functions usually entails sacrifices in accuracy and/or performance. This articles combines several available techniques to obtain implementations of double-precision interval elementary functions that are as accurate as mathematically possible, less than twice slower than their best available scalar counterparts, and fully verified. Implementations of exponential and natural logarithm on an Itanium® 2-based platform support these claims.
Keywords: Interval arithmetic, floating-point, elementary functions.

1 Introduction

The IEEE-754/IEC 60559 standard for floating-point arithmetic[3] defines floating-point formats, among which single and double precision, and specifies the behaviour of basic operators ($+, -, \times, \div, \sqrt{\ }$) in four rounding modes (round to the nearest or RN, towards $+\infty$ or RU, towards $-\infty$ or RD and towards 0 or RZ). The *directed* rounding modes RU, RD and RZ are intended to help implementing interval arithmetic (IA). Most processors now provide hardware support for the IEEE-754 standard, and thus the opportunity for hardware-accelerated interval arithmetic.

However, until recently, the limitations of this hardware support on most processors meant that interval arithmetic was much slower than one would think just by counting the operations. In particular, frequently changing the rounding mode had a large overhead, because processors needed to flush their FP pipeline at each mode change. This is no longer a problem on recent processors like the Intel® Itanium® processor family [5], and recent UltraSPARC® III processors with the VIS™ instructions set extension by Sun Microsystems [2].

Among the next useful building blocks for efficient FP IA support is the elementary function library. This articles demonstrates the feasibility of a “perfect” elementary function interval library in IEEE-754 single and double precision. Here, perfect means

1. returning *sharp* intervals, *i.e.* intervals that are as tight as the representation mathematically allows,
2. with verified algorithms and implementations, and
3. with performance within a factor 2 of the best available corresponding scalar elementary functions.

To our knowledge, among existing libraries offering double-precision interval elementary functions (reviewed in Section 2), none offers this combination of features.

The main idea is that it takes very little to convert a *proven correctly rounded elementary function*, as developed in the `crlibm` project [1, 6], into a perfect interval function. More specifically, all the required software building blocks are already in place in the `crlibm` framework, and the proof of the containment property for the interval function can be derived straightforwardly from the proof of the correct rounding property for the function, which is available in `crlibm`. These ideas and techniques are developed in Section 2, and Section 3 discusses more specifically performance and optimisation issues, and presents some experiments on an Itanium-2 based platform. The conclusion includes a discussion on the pros and cons of sharp intervals.

2 From correctly rounded to interval functions

2.1 Floating-point elementary functions

The IEEE-754 standard requires correct rounding for $+$, $-$, $\times$, $\div$ and $\sqrt{\ }$, but has currently no such requirement for the transcendental functions (this could change in the upcoming revision of the standard). The main reason for this is the *table maker's dilemma* (TMD): FP numbers are rational numbers, and in most cases, the image $\hat{y}$ of a rational number x by an elementary function f is not a rational number, and can therefore not be represented exactly as an FP number. The *correctly rounded* result will be the floating-point number that is closest to this mathematical value (or immediately above or immediately below, depending on the rounding mode). A computer will evaluate an approximation y to the real number $\hat{y}$ with accuracy $\overline{\varepsilon}$, meaning that the real value $\hat{y}$ belongs to the interval $[y/(1+\overline{\varepsilon}), y/(1-\overline{\varepsilon})]$. The dilemma (named in reference to the early builders of logarithm tables) occurs when this information is not enough to decide correct rounding. For instance, if $[y/(1+\overline{\varepsilon}), y/(1-\overline{\varepsilon})]$ contains a floating-point number z, it is impossible to decide the rounding up of $\hat{y}$: it could be z, or its successor.

A technique for computing the correctly rounded value, published by Ziv [24] and first implemented in IBM Accurate Portable Mathlib, is to improve the accuracy $\overline{\varepsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, a first, quick approximation y_1 to the value of $f(x)$ is evaluated, with accuracy $\overline{\varepsilon}_1$. Knowing $\overline{\varepsilon}_1$ and therefore $[y_1/(1+\overline{\varepsilon}_1), y_1/(1-\overline{\varepsilon}_1)]$, it is possible to decide if rounding y_1 is equivalent to rounding $y = f(x)$, or if more accuracy is required. In the latter case, the computation is restarted using a (slower) approximation of accuracy $\overline{\varepsilon}_2$ better than $\overline{\varepsilon}_1$, and so on. This approach leads to good average performance, as the slower steps are rarely taken. Besides, if the worst-case accuracy required to round correctly a function is known [14], then only two steps are needed. This improvement, implemented in the `crlibm` project [1, 6], makes it easier to optimise and prove each step. Here the proof is mostly a computation, for each step, of a sharp bound $\overline{\varepsilon}$ on the total error of the implementation (due to accumulated approximation and rounding errors). In the first step, this bound is used to decide whether to go for the second step. In the second step, it is enough that this bound is lower than the worst-case accuracy to guarantee correct rounding. Note that this technique works for obtaining correctly rounded values for any of the IEEE-754 rounding modes.

2.2 FP interval elementary functions

The containment property for elementary functions typically requires the computation of the images of both ends of the input interval, one with rounding up, the other with rounding down, depending on the monotonicity of the function. For functions which are not monotonic on their input interval, the situation may be more complex. However, elementary functions, by definition, have useful mathematical properties which can be exploited. For instance, scalar implementations of periodic functions always begin with a range reduction which brings the argument in a selected period. From the information computed in this range reduction, one may deduce the monotonicity information needed to manage interval functions [18].

2.3 A matter of proof

For intervals to guarantee *validated* numerics, it is crucial to provide an extensive proof of the containment property, which again boils down to the proof of an error bound. The best examples so far are the proof [12] of the elementary functions of C-XSC/`fi_lib`/`filib++`[3], and that of `crlibm` [1]. The first implement error computation using their own interval arithmetic package, the second use Maple and, more recently, the Gappa proof assistant [7]. The main difference is that the errors computed for `crlibm` are much tighter (in the order of 2^{-60}, versus 2^{-51} to 2^{-53} for the `fi_lib` family).

Both proofs rely on external tools to compute the approximation error of a Remez polynomial, which may be considered a weakness in the proof as these tools are not proven themselves. Harrison [11] computes a machine-verified bound of such approximation error in two step, first approximating the exponential with a suitable Taylor series, then proving a bound on the distance between the Remez and the Taylor polynomials – a tractable but non trivial task. An easier (but less efficient) approach is used by Ershov and Kashevarova [9] for the mathematical library of LEDAS Math solver[4]. They directly evaluate the function using Taylor or Chebychev series whose coefficients have been rounded in the proper direction. A detailed proof of the implementation is missing, but [9] is a convincing sketch.

Another approach was used by Rump for INTLAB [20] to capitalise on existing `libm` implementations, which are highly optimised and accurate. Rum's algorithm computes the value of the function as a small deviation (given by a small polynomial) with respect to some reference point (computed by zeroing the least significant bits of the mantissa of the input number). The maximum error due to this polynomial is easy to bound. The algorithm then adds the deviation to the value of the function at the reference point which is given by a call to the standard `libm`. The maximum error of the `libm` function with respect to the true value has been precomputed beforehand (only once) over all reference points, which allows to bound the total error of this evaluation scheme. With respect to an implementation of the function from scratch, this scheme is a compromise between performance and ease of proof: The code will be slower and the result less accurate, because it adds computations to those performed by the `libm`. However, the proof will be much easier (the evaluation error of a small polynomial). However, comparing Rump's proofs [20] and the *from scratch* approaches [9, 12, 7] seems to indicate that this approach saves less than half the proof work.

To our knowledge, there is no published proof of the implementations of elementary functions of other leading FP IA packages. We assume that the interval functions in Sun development environments derive from those published by Priest [18]. PROFIL/BIAS[5] uses calls to the standard mathematical library with safety margins added. This does not provide a vali-

[3] `http://www-info2.informatik.uni-wuerzburg.de/staff/wvg/Public/filib++.html`
[4] `www.ledas.com`
[5] `http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html`

dated solution: Standard libraries are not proven and current standards do not even specify their accuracy.

2.4 Accuracy versus performance

From the *accuracy* point of view, the libraries by Priest [18] and Ershov and Kashevarova [9] will return a sharp interval with good probability, otherwise the interval will lose at most one ulp on each bound. In contrast, INTLAB, the `fi_lib` family and PROFIL/BIAS will almost always return an interval where each bound is several ulps away from the ideal one. The reason for this loss of accuracy is the use of the target precision for the intermediate computations, which entails that several ulps are lost in the process (to rounding or approximation errors). Priest [18] uses classical table-based techniques[22, 23, 15] to reach an accuracy better than 1.5 ulps. There is a trade-off between accuracy and performance here: Priest mentions earlier works [4, 13] which come closer to 1-ulp accuracy at the expense of performance, as a larger working precision has to be emulated.

From the *performance* point of view, all the libraries mentioned have performance within a factor 20 of the `libm`, their scalar equivalent. The most efficient library by far is that of Priest, who remarked [18] that a monolithic interval function should be faster than two successive calls to the scalar function: The computation of the two endpoints is intrinsically parallel, allowing for more efficient use of modern super-scalar, deeply pipelined processors than two successive calls to one function. Besides, computing both endpoints in parallel allows to share some of the burden of memory accesses for table look-ups.

2.5 Techniques for the perfect interval library

To sum up the previous discussion, a perfect interval library may be obtained by

1. parallel evaluation of both endpoints to ensure efficient pipeline usage, as suggested by Priest,
2. a two-step approach *à la* Ziv for each endpoint to ensure sharpness, and nevertheless performance,
3. and a comprehensive proof as first shown in `fi_lib`, but using the automated and performance-driven techniques of the `crlibm` framework.

A difficulty is that, as far as performance is concerned, points 1 and 2 conflict: The two-step approach means tests and branches, which have to be laid out carefully in order to allow a streamlined parallel execution. This question is studied in Section 3.

2.6 When worst cases are missing

It is actually easier to write a perfect interval function than a correctly rounded scalar one. Indeed, the worst-case accuracy required for correct rounding is still missing for many functions on many intervals[6]. For some intervals and some functions (most notably the trigonometric functions for large arguments), it is known that current techniques for computing worst case accuracy will not work [14]. This prevents us from writing a two-step proven correctly rounded function, however we may still design an interval function which will be verified, efficient, and tight with a very high probability. The idea is to test, at the end of the second step, if any of

[6] `http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm`

the interval bounds being computed is difficult to round, and to enlarge the returned interval by one ulp only in this case.

Will such a case happen? According to statistical arguments by Gal [10, 17], the worst case accuracy required for correct rounding the trigonometric functions, on the domain where worst cases are missing, is expected to be about 2^{-117}. Current `crlibm` code for these functions is accurate to 2^{-124}. Therefore, the probability that there exist a floating-point input argument for which the previous test would succeed (and require to enlarge the interval) is roughly of 2^{-7}. In other words, a `crlibm`-based implementation will be *probably perfect*. In any case, the numerical effect of this possible enlargement can be disregarded, and the containment property will provably not be violated.

Another option would be to launch a higher precision computations, should such a case happen, but it is difficult to justify writing, proving and maintaining code that is very probably useless.

Note that the example functions presented in the sequel do not have this problem, since their worst case accuracy is known. Note also that in the example of the trigonometric of a large argument, a quick test following argument reduction may determine if the input interval contains a period of the function, in which case the output interval is simply $[-1, 1]$. Here the cases where we are currently unable to compute the worst-case accuracy are also the cases where the interval function is most likely to be degenerate, and this is not a coincidence.

From a performance point of view, this additional test may be fairly expensive: One needs to consider all the bits between the 53rd (round bit) and the last significant bit (the 124th in current implementation), and check whether they are all zeroes or all ones. However, this test will only happen in the second step, therefore its average performance impact will be negligible [6].

3 Experiments and results

In this section, an Itanium-2 based platform was chosen because it provides the best hardware support for interval arithmetic among currently available processors. The exponential and logarithm functions were chosen because their worst-case accuracy for correct rounding in double precision is known, and because scalar implementations optimised for the Itanium-2 processor of such correctly rounded functions were already available [6, 8].

3.1 An exponential and a logarithm function

Space prevents us to detail the algorithms used to evaluate the functions. They are exactly the same as their scalar counterparts in the `crlibm` distribution using double-extended arithmetic [6, 1], which itself was inspired by previous work [21, 16, 15]. We concentrate here on specific work for turning these scalar functions into interval ones.

Both functions are implemented in two Ziv steps: the first step handles exceptional cases (zeroes, sub-normals, infinities) and evaluates in parallel the value of the function at both endpoints, using double-extended precision (64 bits of mantissa) with round to nearest mode. As some accuracy is lost to the rounding error, these first steps are accurate to $\overline{\varepsilon} = 2^{-61}$ for both functions (a comprehensive proof of this error bound is also available in the `crlibm` CVS repository). This means that correct rounding can be deduced from this intermediate result with a probability of about $1 - 2^{-61+53+1}$, or in more than 99% of the cases.

3.2 Directed rounding on the Itanium

The Itanium processors can mix different rounding modes and precisions at no cost thanks to the availability of 4 *floating-point status registers* (FPSR), selected on a per-instruction basis [5]. Out of these, SF0 is usually preset to RN in double, and SF1 to RN in double-extended. We chose to use SF3, preset as "round up to double precision", to implement directed rounding. Round down is obtained by rounding up the opposite, in order to use only one of the extra FPSR (another possible use for these registers is speculation). Figure 1 gives some macros, using assembler intrinsics accepted by the Intel and HP C compilers, that implement round up and down of a double-extended number (for the first step) and a double-double-extended number (or DDE, for the second step) [6]. These macros use *fused multiply and add* (FMA) operations, which compute $a * b + c$ with only one rounding.

3.3 Parallel evaluation

Writing two parallel paths is straightforward. Figure 2 gives the code of the polynomial evaluation of the logarithm according to Estrin's scheme [17]. Suffixes `i` and `s` denote variables corresponding to the the lower and upper bounds of the interval. All the variables are double-extended.

The Itanium-2 processor provides two parallel FMAs with 4-cycle latency. Estrin's scheme is particularly suited to such a processor [21, 5] because it exposes parallelism which improves pipeline usage. Running two evaluations in parallel improves pipeline efficiency further.

Obviously, such a scheme will require twice as many registers as a sequential one. This is not a problem on Itanium, however. Ideally, this should not be our concern: a programmer should expose as much parallelism as possible, and leave it to the compiler to exploit it or not, depending on the capabilities of the target processor.

A reviewer remarked that there is a lot of redundant work in these parallel evaluations as soon as the input interval is tight enough. Unfortunately, we see no way of exploiting that to improve code efficiency: It would require to add tests to the code, which will slow down the average case. We considered testing the special case of a point interval, as is done in `fi_lib`, and even there felt that the benefit (less than 50% improvement for point intervals) was not worth the overhead to the general case (about ten cycles).

3.4 Combined rounding test

As multiple dependent tests may be expensive in a deep pipeline, both tests for correct rounding are merged into one single test. These tests consist in extracting the bits of the mantissa between the 53rd (round bit for double-precision) and the 61st (last significant bit considering the accuracy of the computation), and checking whether they are either all zeroes or all ones, which would mean that the number is very close to a double-precision number [14].

```
#define ROUND_EXT_TO_DOUBLE_UP(x)  _Asm_fma(_FR_D, 1.0, x, 0.0, _SF3)      // 1*x + 0
#define ROUND_EXT_TO_DOUBLE_DOWN(x)  -_Asm_fma(_FR_D, -1.0, x, 0.0, _SF3)
#define ROUND_DDE_TO_DOUBLE_UP(xh,xl)   _Asm_fma(_FR_D, 1.0, xh, xl, _SF3) // 1*xh + xl
#define ROUND_DDE_TO_DOUBLE_DOWN(xh,xl) -_Asm_fma(_FR_D, -1.0, xh, -xl, _SF3)
```

Fig. 1. Rounding macros for Itanium

```
z2i = zi*zi;                z2s = zs*zs;
p67i = c6 + zi*c7;          p67s = c6 + zs*c7;
p45i = c4 + zi*c5;          p45s = c4 + zs*c5;
p23i = c2 + zi*c3;          p23s = c2 + zs*c3;
p01i = logirhi + zi*c1;     p01s = logirhs + zs*c1;
z4i  = z2i*z2i;             z4s  = z2s*z2s;
p47i = p45i + z2i*p67i;     p47s = p45s + z2s*p67s;
p03i = p01i + z2i*p23i;     p03s = p01s + z2s*p23s;
p07i = p03i + z4i*p47i;     p07s = p03s + z4s*p47s;
logi = p07i + Ei*log2h;     logs = p07s + Es*log2h;
```

Fig. 2. Parallel polynomial evaluation

Other possibilities exist for this test, but this approach, using only 64-bit integer arithmetic which is very fast on the Itanium, was found the most effective.

Figure 3 gives the corresponding code for the logarithm. Most variables are 64-bit integers, except `logi` and `logs` which are double-extended, and `result` which is an interval of doubles. The `GET_EXT_MANTISSA` macro extracts the 64-bit mantissa of a double-extended number (its implementation is processor-dependent). The `log_accurate` function performs the second step in double-double-extended arithmetic, and returns a pair of double-extended numbers which are then added together to a double with rounding up.

```
/* Tentatively set the result */
result.INF = ROUND_EXT_TO_DOUBLE_DOWN(logi);
result.SUP = ROUND_EXT_TO_DOUBLE_UP(logs);

/* Now check correct rounding using 64-bit arithmetic on the mantissas  */

mantissai = GET_EXT_MANTISSA(logi);
mantissas = GET_EXT_MANTISSA(logs);
bitsi =  mantissai & (0x7ff&(accuracymask));
bitss =  mantissas & (0x7ff&(accuracymask));
infDone= (bitsi!=0) && (bitsi!=(0x7ff&(accuracymask)));
supDone= (bitss!=0) && (bitss!=(0x7ff&(accuracymask)));

/* Only one test, expected true */
if(__builtin_expect(infDone && supDone, TRUE))
  return result;

/* otherwise lauch accurate computation */
if(!infDone) {
  log_accurate(&th, &tl, zi, Ei, indexi);
  result.INF = ROUND_DDE_TO_DOUBLE_DOWN(th,tl);
}
if(!supDone) {
  log_accurate(&th, &tl, zs, Es, indexs);
  result.SUP = ROUND_DDE_TO_DOUBLE_UP(th,tl);
}
```

Fig. 3. Combined rounding test

Exponential is similar, with the exception of a trick that makes it slightly less readable: The last operation of the chosen algorithm is a multiplication by a power of two. This operation being exact in IEEE-754 arithmetic, the rounding test can be performed on the intermediate result before this last multiplication, so that it runs in parallel with it.

3.5 A note about portability

The code of Figures 2 and 3, as well as the code of the `log_accurate` function, no longer contains any Itanium-specific code: to compile it on an x86-compatible processor, all it takes is redefine implementations of the macros. The proof depends on lemmas that describe the behaviour of the macros, so these lemma also have to be re-proven, which is very simple here. The rest is 64-bit integer arithmetic which is supported by compilers on 32-bit machines, although possibly in a sub-optimal way. Previous experience with the scalar logarithm of `crlibm`[7] suggests that with such a macro-based approach, portability, provability and efficiency do not necessarily conflict.

3.6 Accuracy and performance

Absolute performance results are given in Table 1. One call to our interval function is faster than two calls to the corresponding point functions with directed rounding, as predicted by Priest [18]. The proposed implementation is comparable to twice the default scalar `libm` (which does not pay the price of a function call since it is inlined by the compiler).

Table 2 evaluates the performance and accuracy of the following iteration:

$$\begin{cases} y_{n+1} = \log(x_n) \\ x_{n+1} = e^{y_{n+1}} \end{cases} \tag{1}$$

It is easy to check that the initial interval size should grow by exactly two ulps at each iteration, in the case of a sharp implementation. This expected behaviour is observed on our implementation. The growth of 13 ulps per iteration of the `fi_lib`-based implementations is consistent with [12].

Library	exp	interval exp	log	interval log
`libm`	42	n/a	31	n/a
`fi_lib`	586	1038	619	1158
`crlibm`	60	69	66	96

Table 1. Compared performance results on Itanium 2 processor (timings in cycles).

Experimenting around this code, we found that the data structure used for intervals in `fi_lib` has a significant overhead, which explains that the timing of a loop (which also performs some bookkeeping) is much higher than the sum of the timings of the functions in Table 1.

Library	Cycles	Ulps
`fi_lib`	2215	13
`crlibm`, RU and RD functions	342	2
`crlibm` merged interval function	249	2

Table 2. Average performance in cycles and interval growth in ulps of iteration (1) on Itanium 2

[7] See the `log-de.c` file in the `crlibm` distribution since version 0.10beta. The macros are defined in `double-extended.h`

4 Concluding remarks

This article surveyed and demonstrated software techniques available for the implementation of "perfect" interval floating-point elementary functions, where perfect means tight, efficient and fully verified. It also remarked that it is possible to write a probably-perfect interval function even in the cases where theoretical results are missing to write a proven correctly rounded function. In such cases, the function will still be efficient and verified, but might very rarely (and probably never) return an interval that is not the tightest possible.

As the presented implementations are sharp, there can be no further improvement in accuracy. Performance, however, can still be improved. The main issue is the organisation of the tests for special cases and correct rounding. Present code is slow when handling intervals where a bound is infinity or zero, for instance. Whether this is a problem will depend on the application. In principle, infinities occur often in the development of an IA application, but should occur rarely in production.

The functions presented in this paper are still experimental and non-portable, and a lot of work will be needed to develop a full portable library (needed to compare to Sun's current offering, for example). The `crlibm` framework is striving to encapsulate processor-specific optimisations into common C macros selected at compile time. A conclusion of the present study is that interval functions can build upon this framework and extend it.

Revol and Rouillier [19] and Kahan[8] have advocated the use of multiple-precision interval arithmetic. The idea is to perform the bulk of computations in native precision, and increase the precision only when needed due to interval bloat. A perfect interval library as described in this article is a building block for implementing such efficient strategies.

In defence of sharp bounds

As a conclusion, one might discuss the usefulness of providing sharp bounds in interval elementary functions.

On one side, numerical portability of an application (the fact that it will return the same answer whatever the system) is probably not so much of an issue as soon as interval arithmetic is used: the result will be validated nevertheless. Our sharp library will from time to time return an interval smaller by one or two ulps than Priest's or Ershov's. From a practical point of view, this accuracy improvement is probably of little significance. An interval library implemented with the accuracy standards of current scalar `libms` [21, 15] would typically bloat the intervals by one ulp in one percent of cases only, and have even better performance than the present approach.

On the other side, the points in favour of sharp bounds are the following:

– It is more elegant to always return the best possible interval considering the number representation.
– It is consistent with the existing IEEE-754 standard.
– It allows for fair benchmarking where one compares functionally equivalent programs.
– It entails small performance penalty in average.

Does it entail more development work? Not really, since even a non-sharp interval function needs a proof of its error bounds. Experiments conducted in `crlibm` consistently show [6] that

[8] `www.cs.berkeley.edu/~wkahan/Mindless.pdf`

the proof of the second step is much easier because it doesn't include the argument reduction (already proven in the first step), and it needn't be optimised as tightly as the first step.

Thus, the cost of sharp bounds is an increase in code size, but negligible performance overhead and much less than doubling the proof effort. The benefits, small as they may be, may be worth the effort.

References

1. CR-Libm, a library of correctly rounded elementary functions in double-precision. `http://lipforge.ens-lyon.fr/www/crlibm/`.
2. Interval arithmetic in high performance technical computing. Technical report, Sun Microsystems, September 2002.
3. ANSI/IEEE. *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559).* 1985.
4. K. Braune. Standard functions for real and complex point and interval arguments with dynamic accuracy. In *Scientific Computation with automatic result verification*, pages 159–184. Springer-Verlag, 1988.
5. M. Cornea, J. Harrison, and P. T. P Tang. *Scientific Computing on Itanium-based Systems.* Intel Press, 2002.
6. F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th Symposium on Computer Arithmetic*, pages 288–295. IEEE Computer Society Press, June 2005.
7. F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions. Technical Report RR2005-43, LIP, September 2005.
8. F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 2006. To appear.
9. A. G. Ershov and T. P. Kashevarova. Interval mathematical library based on Chebyshev and Taylor series expansion. *Reliable Computing*, 11(5):359–367, 2005.
10. S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.
11. J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.
12. W. Hofschuster and W. Krämer. FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Technical Report Nr. 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1998.
13. W. Krämer. Inverse standard functions for real and complex point and interval arguments with dynamic accuracy. In *Scientific Computation with automatic result verification*, pages 185–211. Springer-Verlag, 1988.
14. V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.
15. R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, April 2001.
16. P. Markstein. *IA-64 and Elementary Functions: Speed and Precision.* Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.
17. J.-M. Muller. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997/2005.
18. D. M. Priest. Fast table-driven algorithms for interval elementary functions. In *13th IEEE Symposium on Computer Arithmetic*, pages 168–174. IEEE, 1997.
19. N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. In *Workshop on Validated Computing*, pages 155–161. SIAM, 2002.
20. S. M. Rump. Rigorous and portable standard functions. *BIT Numerical Mathematics*, 41(3), 2001.
21. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In *14th IEEE Symposium on Computer Arithmetic*, pages 4—11. IEEE, April 1999.
22. P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.
23. P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, December 1990.
24. A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.

Accurate Math Functions on the Intel IA-32 Architecture: A Performance-Driven Design

Cristina Anderson, Nikita Astafiev, and Shane Story

Intel Corporation
`{cristina.s.anderson,nikita.astafiev,shane.story}@intel.com`

Abstract. We discuss our design principles and techniques for highly accurate, ISO C99-compliant math library routines on the Intel IA-32 architecture. Special focus is placed on single and double precision functions, which draw benefit from the SSE and SSE2 extensions, while having to overcome some challenges related to achieving the desired accuracy.

1 Introduction

High performance, accurate math functionality is important for many modern applications. The Intel IA-32 architecture is the most ubiquitous computing platform today, found in anything from compute-intensive systems for scientific applications to servers, desktops, notebooks and even game consoles.

Our goal is to provide very good performance and accuracy for all the different Intel IA-32 processors. Our math library routines are designed with a threshold of 0.55 ulp (units-in-the-last-place) for the maximum absolute error in round-to-nearest mode. A number of key routines that are most frequently used in applications are finely tuned for close to optimal performance within our design specifications.

Microarchitectural differences between IA-32 processors and architecture extensions such as new instructions sometimes require us to maintain different versions of performance-critical routines, so that the best available implementation is used on each processor. The library makes use of a special dispatch mechanism to select the appropriate implementation for each processor version.

The Intel math library is provided with the Intel IA-32 compiler and supports the full set of ISO C99 [1] functions in single, double, double extended and quad precision, as well as a number of functions needed by specialized applications, e.g. financial applications.

Besides our high accuracy goal, we ensure the correct setting of the overflow, underflow, divide-by-zero and invalid hardware exception flags, as well as correct behavior when these exceptions are enabled. Due to both practical considerations and architectural limitations we do not require the Inexact flag to be set correctly in most routines. For the same reasons, the architecture-defined Denormal flag is also set incorrectly in some implementations.

Much of our paper will concentrate on algorithm design for the SSE/SSE2/SSE3 extensions to the IA-32 instruction set, since the majority of our optimized single and double precision routines are based on these instructions. Subsection 2.3 offers a brief introduction to the SSE/SSE2/SSE3 instructions. Algorithm design for this instruction set is more complex than x87-based design because of the lack of extended precision, and also because the x87 unit offers support instructions for the most common functions.

Section 2 gives a brief presentation of IA-32 architectural details and other general issues that are relevant to our performance-centered design. Section 3 shows how SSE/SSE2 code can be improved in some situations, by replacing slower operations with faster instruction sequences. Section 4 discusses ways to overcome the lack of additional precision in intermediate steps for double precision routines implemented with the SSE2 instruction set. Section 5 illustrates our design methods with a discussion of two SSE2-based implementations. Section 6 presents our design principles for handling special cases. Section 7 presents some accuracy and performance data for our library and two other publicly available libraries.

2 Overview of the Architecture and General Optimization Techniques

This section covers aspects of the IA-32 architecture that relate to our floating-point function design, and general coding guidelines for achieving better performance. For more details see [2], [3], [4]. We also summarize the assembly instructions most frequently used in our hand-optimized code in Table 1.

2.1 General Purpose Instructions

The IA-32 architecture offers eight 32-bit general purpose registers (GPRs). There are sixteen 64-bit registers when the 64-bit extensions are enabled.

The basic integer arithmetic and logical instructions are very fast; depending on the processor, they execute in 1 or just 0.5 clock ticks. Examples of instructions in this category are ADD/SUB, AND, OR, NOT, CMP. Shift instructions are also reasonably fast. We use these instructions frequently for table index generation and for filtering out special cases (together with conditional branch instructions).

Other general purpose instructions that are used only occasionally due to their longer latencies include integer multiply, conditional moves, and bit scan instructions. Integer divides are avoided due to their high latency.

When 64-bit extensions are turned on, the larger width of general purpose registers makes these fast integer operations more attractive for applying the scaled integer techniques outlined in [5]. The IA-32 architecture offers good performance floating-point primitives, however. The cost of data transfers between floating-point and general purpose registers is also relatively high; such transfers involve either memory transactions, or data manipulation equivalent to at least one floating-point latency. As such, integer-based implementations of floating-point math functions are typically not the best choice on IA-32 and IA-32 with 64-bit extensions.

When optimizing for specific microarchitectures, one needs to be aware of and avoid situations that lead to unwanted stalls. One such example is the artificial dependency that is created when a partial register write is followed by a full register read on the Intel® Pentium® M, e.g. when a 16-bit load to AX is followed by a 32-bit read of EAX. This problem can be easily eliminated by using full register updates.

Software calling conventions specify only three scratch registers in the standard 32-bit mode. To use any of the other general purpose registers, a routine must incur the cost of saving and restoring them. (There are at least seven scratch registers in 64-bit mode, depending on the environment.)

2.2 x87 Floating-Point Instructions

The x87 floating-point unit has eight 80-bit data registers, organized as a stack. This execution environment is fully IEEE 754 compliant [6], and thus supports the single, double, double-extended precisions and all IEEE rounding modes.

Besides the basic arithmetic instructions (add, subtract, multiply, divide, square root), x87 provides support instructions for elementary functions such as the exp, log and trigonometric families, as well as support for the remainder operations.

x87-based implementations remain the first choice for double-extended precision functions.

The more versatile SSE/SSE2 extensions offer a better alternative for most single and double precision functions. For a few functions x87 implementations are still preferred, for example the IEEE remainder and hypot.

2.3 SSE, SSE2 and SSE3 Instructions

The SSE extensions include a set of 128-bit data registers called XMM registers (eight registers in 32-bit mode, sixteen in 64-bit mode), 4-way SIMD (single-instruction operating on multiple-data sets) and scalar single precision instructions, and 64-bit integer instructions on packed operands. The SSE2 extensions add double precision instructions in scalar and packed versions (2-way SIMD), as well as 128-bit SIMD integer instructions (operating in parallel on 2 64-bit, 4 32-bit, 8 16-bit, or 16 8-bit integers). The SSE3 extensions add a small number of instructions to the instruction set.

An interesting feature for our design purposes is the ability to treat data in XMM registers as either floating-point or integer. Our implementations often require changing the sign or exponent of the arguments, or separating the leading several bits from the mantissa. This can be quite easily accomplished by manipulating the argument in floating-point format with SSE2 integer and logical operations such as AND/OR/XOR, logical shifts and an occasional integer add/subtract.

SIMD operations are useful for boosting performance whenever the same operations need to be applied to independent sets of data. Our implementations of scalar math functions use SIMD most often to efficiently evaluate a polynomial correction to the initial approximation. Another example is the case of the hyperbolic functions *sinh* and *cosh*; *exp(x)* and *exp(-x)* can be computed at the same time with the help of SIMD instructions.

To achieve the best performance, our implementations avoid the use of high latency instructions and replace them with faster code sequences whenever possible. These aspects will be discussed in more detail in the following sections.

The standard calling convention in 32-bit mode requires the floating-point arguments to be passed on the memory stack and the floating-point result to be returned on the x87 register stack. This means our optimized SSE/SSE2 implementations must incur the overhead of loading the arguments from memory, then transferring the result from XMM to the x87 stack (through memory). In rare instances this additional overhead tips the balance in favor of an x87 implementation. Newer calling conventions for single and double-precision pass the arguments and return the result in XMM registers.

It is important to avoid known coding pitfalls that can degrade performance on some processors. One example here is a small, unaligned load shortly following a large store, which results in a significant stall on the Intel® Pentium® 4. This undesirable situation may occur when a small load is used to access the exponent bits of a double precision argument at the beginning of the routine, after the argument has just been stored on the stack. A full load of the 64-bit argument should be done instead.

Table 1 shows a list of the IA-32 instructions that are used in most of our hand-optimized SSE/SSE2 implementations. Some other instructions are needed on a more occasional basis. The complete list of IA-32 instructions is available in the references.

General Purpose		
Arithmetic	ADD, SUB	
Logical	OR, AND, XOR	
Shifts	SHL, SHR, SAR	
Tests	CMP, TEST	
Loads	MOV	
x87		
Load	FLD	
SSE/SSE2/SSE3	**Double Precision**	**Single Precision**
FP SIMD/scalar add, subtract	ADDPD/SD, SUBPD/SD	ADDPS/SS, SUBPS/SS
FP SIMD/scalar multiply	MULPD/SD	MULPS/SS
Logical	ORPD, ANDPD, XORPD	ORPS, ANDPS, XORPS
Load (one element)	MOVSD	MOVSS
128-bit loads	MOVAPD	MOVAPS
Pack	MOVDDUP	
Pack/unpack	PSHUFD	
Transfers to/from GPR	MOVD, PEXTRW, PINSRW	
Format conversions	CVTSS2SD, CVTSD2SS	
Logical shifts	PSRLQ, PSLLQ, PSRLD, PSLLD	

Table 1. Instructions Frequently Used in Optimized Single and Double Precision Routines

3 Speeding Up SSE/SSE2/SSE3 Implementations

As mentioned in the previous section, it is often necessary to consider and work around microarchitectural details in order to obtain close to optimal performance. Instructions such as divide and square root have higher latency on all processor versions, thus their use should be limited. In addition, some higher latency instructions can often be replaced by faster code sequences. Some of the techniques we use are presented in this section. The code examples given are in IA-32 assembly language when needed for clarity (if the data is manipulated by instructions of different types, e.g. floating-point data manipulated with integer instructions), and otherwise in C pseudo-code for better readability.

3.1 Fast Single-to-Double Convert

On those processors that have a slow CVTSS2SD (convert-single-to-double) instruction, the following sequence may work better for an input $x > 0$ (x stored in xmm0):

```
psllq   xmm0,  52 - 23     //
paddd   xmm0,  xmm1        // xmm1 contains the constant 0x3800000000000000
                           // now xmm0 contains x in double precision
```

3.2 Fast Convert-to-Integer

The fractional bits of a floating-point value x stored in an XMM register can be shifted out when a sufficiently large constant is added. The integer part of x (rounded according to the current rounding mode) is then obtained by subtracting the large constant from the previous result. The latency is twice that of a floating-point add (if the integer is needed in an XMM register), or the latency of an FP add plus the latency of a transfer to GPR (if the result is needed in a general purpose register). The SSE2 instruction set provides convert-to-integer instructions; however, their latencies are typically higher. Here is an example of convert-to-integer without the use of convert instructions. The argument x is assumed to be in the range $|x| < 2^{51}$, and is usually much smaller ($|x| < 2^{31}$).

Let $S = 1.5 \cdot 2^{52}$ (double precision), or $S = 1.5 \cdot 2^{23}$ (single precision). Assume x is stored in xmm0, S is stored in xmm1. Then

```
addsd/ss xmm0, xmm1   //
movd     eax,  xmm0   // if $|x| < 2^{31}$, then eax holds int($x$)
subsd/ss xmm0, xmm1   // xmm0 now holds int($x$)
```

Note that the same technique can be used to retain the desired number of fractional bits, or to shift out the less significant integer bits of x, if needed. For example, to round double precision x to 5 fractional bits, use $S = 1.5 \cdot 2^{52-5}$ (assuming $|x| < 2^{51-5}$).

3.3 Fast Multiplication by Powers of 2

Multiplication by powers of 2 only modifies the exponent of a floating-point number. It is possible in some cases to treat the floating-point value stored in an XMM register as an integer and add the appropriate integer constant to it, such that the result interpreted as a floating-point number is $x \cdot 2^k$. The exceptions are special cases such as overflow/underflow, or arguments with special encodings (e.g. Infinity, denormals). The algorithm designer can make sure that these cases do not occur in the main computation path, though.

This programming trick accomplishes multiplication in one PADDD latency instead of a full FP multiply latency, which is typically 2 to 3 times slower.

Note that while this technique worked well on the Intel® Pentium® 4 processors, it may not be a fast solution on some of the newer Intel® Pentium® M cores. (Moving data between the integer and FP execution stacks incurs a small penalty on some microarchitectures.)

3.4 Branch Collapsing

This technique is useful for minimizing the number of branch instructions that filter special cases out of the main path.

Two or more ranges of special arguments can be combined to use the same test and branch to a special path. For example the tests for $x < a$ and $x > b$ can be rewritten as $x - a < 0$, $x - a > b - a$. Now if a, b, x are integers and $b - a$ is positive and in the signed integer range, an unsigned integer compare can be used for $x - a < 0$. If $x - a < 0$, then its integer representation interpreted as unsigned is larger than $b - a$. The test code would then look as follows:

```
                        // initially eax=$x$, edx=$a$, ecx=$b$
sub   eax, edx          // now eax = $x - a$
sub   ecx, edx          // now ecx = $b - a$
cmp   eax, ecx          //
ja    SPECIAL_PATH      // branch if $x \notin [a, b)$
```

3.5 Conditional Arithmetic Operation

To avoid branch penalties, the following code sequence (using N-bit integers a, b, x, q)

```
if ($a > b$)
   $x$ += $q$;
```

can be replaced by

```
$S \;=\; (b-a) \gg (N-1)$    // sign mask, will be set to -1
$q_1 \;=\; S \,\&\, q$
$x \;+\!= q_1$               // will add 0 if $a \le b$, $q$ otherwise
```

This "trick" can be expanded to replace a sequence such as

```
if ($a > b$)
  <operation 1>
else
  <operation 2>
```

3.6 Fast Absolute Value (Integer Format)

For x an N-bit signed integer ($N = 32$ or $N = 64$, depending on register size):

```
$S = x \gg (N-1)$       // get sign mask with an arithmetic shift right
$y = (x+S)$ ^ $S$       // now $y = |x|$
```

3.7 Reciprocal and Square Root

Whenever the operand is one of a small set of values, it is generally faster to retrieve the full precision result from a table than it is to use the correctly rounded divide or square root instruction. This type of situation can arise when the function is initially approximated based on a few leading bits of the argument.

A lookup table is also a faster solution when less than full precision accuracy is needed. A low accuracy reciprocal is needed to start approximations for functions such as *log* and *cbrt*. One can create a reciprocal lookup table to the desired specifications, or use the RCPSS instruction, which takes a single precision argument and offers about 11.5 bits of accuracy. When load latencies are high, a reciprocal solution based on RCPSS may be faster even for double precision. A quick convert to single precision for input $x \in [1, 2)$ can be performed as

```
psrlq   xmm0, 52-25   // 64-bit shift right
psrld   xmm0, 2       // 32-bit shift right
```

This is then followed by

```
rcpss   xmm0, xmm0    // single precision reciprocal approximation
```

The result can then be processed as desired and converted back to double precision format (see the fast single-to-double convert suggestion above). One could use a similar approach to get a fast square root reciprocal approximation based on the RSQRTSS instruction.

4 Challenges in SSE2 Algorithm Design

When using a sequence of floating-point instructions to compute a result, rounding errors accumulate at each step. When the intermediate results are rounded to the same precision as the final result, the resulting error can easily exceed our goal of at most 0.55 ulp. A 0.55 ulp maximum error goal essentially means that the accumulated errors should be kept below 0.05

ulp, since the final rounding to the desired precision can be as much as 0.5 ulp (in round-to-nearest mode). Obviously, round-to-nearest mode is required for achieving this accuracy goal.

The SSE/SSE2/SSE3 instruction set only offers single and double precision instructions, so special techniques are needed to meet our accuracy requirements for double precision functions.

In many cases the final operation that yields the result is an addition, say $res=T+p$. If T is the dominant term and is an exact value (for example it is a lookup table value), then only the correction term p is affected by approximation error before the final rounding. To meet our accuracy goal, p needs to be at least 32 times smaller in absolute value than T and also computed to sufficient accuracy. This ensures that the maximum error before the final rounding does not exceed 0.05 ulp and with careful design, it is usually not very difficult to achieve.

When computing T^*x in double precision and T is of length $k < 53$ (the length of a double precision mantissa), a more accurate way to perform the operation is $T * x_h + T * x_l$, where x_h is obtained by truncating x to a mantissa of 53-k bits, and $x_l = x - x_h$. $T * x_h$ is exact (the product is no more than 53 bits long), and the only source of error is the lower term $T * x_l$. In order to minimize the final relative error, $T * x_l$ is usually accumulated with other small correction terms before being added to the leading term. Note that a shorter x_h (i.e. a larger k) will lead to larger relative error ($|x_l/x_h|$ is larger). At the same time, the fixed length T is often a starting approximation such as a refined table lookup value. A shorter T (i.e. smaller k) tends to have a performance impact since more work is needed for the correction term. The algorithm designer needs to find a good balance between these issues.

Finally, in some situations higher precision addition or multiplication of two double precision numbers needs to be simulated with the aid of double precision instructions only.

The product $x*y$ can be computed almost exactly as the sum of four products, by splitting the significands of double precision numbers x and y in half: $x = x_h + x_l$, $y = y_h + y_l$. The higher terms x_h, y_h are obtained by truncating x, y to 26 significant bits, so the significands of x_l, y_l are at most 27 bits long. Then $x * y = x_h * y_h + x_h * y_l + x_l * y_h + x_l * y_l$ and only the lower term $x_l * y_l$ is not guaranteed to round exactly, yielding a relative error below 2^{-103} for the full product. It was assumed here that the ranges of x, y are such that none of the four product terms underflow; a modified approach would be required otherwise. As mentioned earlier, the lower terms are accumulated separately and added to an exactly rounded leading term only at the end, in order to minimize the final relative error.

An extended precision sum of two double precision numbers can be expressed exactly as the sum rounded to double precision $S_h = round(a+b)$, and a low part $S_l = a+b-round(a+b)$. In the special case where it is known that $|a| \geq |b|$, the computation sequence is [7]:

```
$S_h = round(a + b)$      // in other words $S_h = a + b_h$
$b_h = S_h - a$
$S_l = b - b_h$
```

5 Implementation Examples

The majority of our optimized algorithms are based on lookup table methods similar to that described in [8]. To illustrate our design techniques, the algorithms for double precision logarithm and single precision cosine functions are discussed below.

5.1 The Double Precision Logarithm Function

The basic identity used in our log computation for argument $x = 2^k * m_x$, $m_x \in [1, 2)$ and k an integer is:

$$log(x) = k * log(2) + log(m_x) = k * log(2) - log(B) + log(B * m_x)$$

Now if $B \approx 1/m_x$, then $log(B * m_x) = log(1 + (B * m_x - 1))$ can be accurately evaluated as a polynomial in $r = B * m_x - 1$.

The special cases are, of course, $x \leq 0$, x is Infinity or NaN, as well as x is a denormal value. While a denormal x is a valid argument to *log*, these arguments would not be processed correctly in the main path (the way it is implemented for maximum performance), and thus need to be directed to a special path. All of these cases are filtered out of the main path as early as possible. The test used merges single interval checks into one compare, in the manner already described in Section 3. The code branches out of the main path if the leading 12 bits in the x representation (sign and exponent), interpreted as an unsigned integer, are either $e_x < 001h$ or $e_x > 7feh$.

In the main path we compute $B \approx 1/m_x$ based on the output of the RCPSS instruction. This is definitely faster than using the DIVSD instruction, and comparable in speed to reading B from a lookup table, since loads from cache memory are still relatively expensive. Using RCPSS also saves storage space for the reciprocal lookup table. B is computed as $nearest\_int(B0 * 2^7 + 0.5)/2^7$, where $B0 \approx 1/m_x$ is the output of RCPSS. This computation is done using SSE logical and integer operations. The double precision representation of x is manipulated to get the single precision representation of m_x (truncated to 24 bits). *B0*=RCPSS(m_x) is then manipulated from single to double precision and rounded to the nearest 1+7 mantissa bits (B). The 7 fractional mantissa bits are transferred to the integer unit, where they are to be used as a table lookup index (j).

The reduced argument $r = B * m_x - 1$ is obtained as $r = (B * m_{high} - 1) + B * m_{low}$, where m_{high} is m_x truncated to 1+7 mantissa bits, $m_{low} = m_x - m_{high}$. Note that both $B * m_{high} - 1$ and $B * m_{low}$ are computed exactly. Also r is guaranteed to be an exact result since $|r|<2^{-7.9}<2^{-7}$ and its representation does not have bits set below 2^{-60}. *log(r)-r* can be estimated to sufficient accuracy by a polynomial of degree 7:

$$p(r) = (c_2 + c_3 * r + c_4 * r^2) * r^2 + (c_5 + c_6 * r + c_7 * r^2) * r^4 * r.$$

The low and high parts of the polynomial are computed in parallel using SSE2 SIMD instructions.

The value of *-log(B)* is read from a 7-bits-in lookup table as (T_{high}, T_{low}), where T_{high}, T_{low} are double precision numbers. The significand of T_{high} is 42 bits long. $log(2)$ is read from memory in the same format: (*L2H, L2L*), where *L2H* also has a significand that is 42 bits long.

The high part of the result is computed exactly as $k*L2H+T_{high}+r_h$, where $k*L2H+T_{high}$ can always be represented in 53 bits since $|k|<2^{11}$ and

$$r_h = round(k * L2H + T_{high} + r) - (k * L2H + T_{high})$$
$$r_l = r - r_h$$

The low part of the result is computed as $r_l + T_{low} + k * L2L + p(r)$. The final operation adds the two parts together.

In the special case of a denormal argument x, its mantissa m_x is normalized and the exponent (k) is adjusted accordingly. After setting the appropriate registers, the code branches back to the main path.

The remaining special cases (x Infinity or NaN or $x \leq 0$) are treated according to the requirements of the standard supported, and the code is rather straightforward to implement. Calls to a special error support function are made to signal an invalid argument ($x \leq 0$).

A quick error analysis shows that the three terms in our basic identity

$$log(x) = k * log(2) + log(m_x) = k * log(2) - log(B) + log(B * m_x)$$

are evaluated with relative errors below $2^{-94}, 2^{-94}$ and 2^{-58}, respectively. This ensures that our accuracy goal is met. Indeed, the maximum error found through extensive testing is well below 0.55 ulp, as shown in the accuracy tables at the end of the paper.

5.2 The Single Precision Cosine Function

The main path of the cosf function works with arguments of absolute value in the range $[2^{-12}, 2^{21})$. Intermediate computation steps take advantage of the higher accuracy provided by double precision instructions. Other values of the argument x are filtered out and sent to special paths.

The quick argument reduction in the main path involves multiplication by a constant L, such that $|L - 2^7/\pi| < 2^{-79}$. The multiplication is performed in two steps: $y = x * L = x * L_{high} + x * L_{low}$, where L_{high} is 30 bits long (leading 30 bits of L) and $L_{low} = L - L_{high}$. Then let $N = nearest\_int(x * L_{high})$, and $r = y - N$ (r is a correction to the reduced argument). Now we define m – an integer, $j \in \{0, 1\}$ and $\alpha \in [0, 2^7)$ such that
$N * \pi/2^7 = (2 * m + j) * \pi + \alpha * \pi/2^7$.

The main identity used in computing the result is:
$\cos(x) = \cos((\alpha + r) * \pi/2^7) = \cos(\alpha * \pi/2^7) * \cos(r * \pi/2^7) - \sin(\alpha * \pi/2^7) * \sin(r * \pi/2^7)$

$\cos(r*\pi/2^7)$ and $\sin(r*\pi/2^7)$ are estimated as polynomials, with relative error below 2^{-30}:
$\cos(r * \pi/2^7) \sim (2 * (2^7/\pi)^2 - r^2) * (\pi/2^7)^2/2$
$\sin(r*\pi/2^7) \sim (r*\pi/2^7)*(6*(2^7/\pi)^2 - r^2)*(\pi/2^7)^2/6 = r*(6*(2^7/\pi)^2 - r^2)*(\pi/2^7)^3/6$

For $\alpha < 2^6$, the double precision $\cos(\alpha*\pi/2^7)$ and $\sin(\alpha*\pi/2^7)$ are retrieved from a 6-bits-in lookup table. These table values are scaled by the appropriate constants as described by the short polynomial expansions above, in order to avoid additional multiplication-by-constant steps. In other words, the result is evaluated as
$\cos(x) = C*(2*(2^7/\pi)^2 - r^2) - S*r*(6*(2^7/\pi)^2 - r^2)$, where C and S are table lookup values.

For $\alpha \geq 2^6$, i.e. $\alpha * \pi/2^7 \geq \pi/2$, we use the lookup table and the identities
$\sin(x) = \sin(\pi - x), \cos(x) = -\cos(\pi - x)$.

Given that the lookup table entries are stored in double precision and the polynomials are evaluated to sufficient accuracy (see above), the maximum error for arguments in the main path is guaranteed to be below 0.55 ulp. This is confirmed by extensive testing (see Section 7).

For arguments $|x| \geq 2^{21}$ a special argument reduction scheme is applied, and the reduced argument is redirected to the main path. We store $T = (2^{8*k}/(2\pi) \mod 1)/2^{8*k-8}$ to about 95 bits accuracy, for $k \in \{0, 1, 2, \ldots, 13\}$. T is stored as three double precision values: $T = T_h + T_m + T_l$. T_h has 1+28 significant bits, T_m has 1+12 significant bits and T_l is a full double precision value. The sizes for T_h, T_m were selected such that multiplication by the single precision argument x is exact, and T_h, T_m can be stored together compactly in 64 bits.

$y = (T_h + T_m + T_l) * x = N + r$ is computed accurately, where
$N = nearest\_int(T_h * x + T_m * x), r = y - N$.

The path for $|x| \in (0, 2^{-12})$ sets the Inexact flag and returns 1. A NaN is returned in special cases (when x is NaN or +/-Infinity).

6 Special Cases

Spurious exceptions and incorrect flag settings other than Inexact (and Denormal in certain special cases) are not acceptable. Spurious Inexact and Denormal are allowed in SSE/SSE2 implementations, unless forbidden by the function specification. For example, *remainder* results are always exact and the IEEE standard requires a correct setting of Inexact; *fmod* results are also always exact, but no standard requires that Inexact be correctly set for *fmod.* In consequence, we may choose a faster algorithm that computes the correct numerical result for *fmod* while setting Inexact, but *remainder* needs to comply with the IEEE standard. So even though the two functions are quite similar, the *remainder* computation may be noticeably slower.

The main reason why we allow Inexact to be set incorrectly is that setting it correctly is usually very expensive in terms of performance, and most users do not check this flag anyway. At the same time, the IA-32 architecture has no hardware support for avoiding exceptions taken in intermediate steps. As an example, the Intel® Itanium® architecture has four status fields that record flag status, so the algorithm designer can use a use a separate status field with exceptions always disabled for those operations that may set spurious flags.

Denormal may be set spuriously when the result of an SSE2 implementation underflows and the software conventions require returning the result on the x87 stack.

The remaining exceptions (Overflow, Underflow, Invalid, Divide-by-zero, as well as Denormal in most cases) occur less frequently in typical computation sequences taking arguments in the normal range. It is usually easy to eliminate the argument ranges that may raise one of these exceptions in an intermediate step. Such ranges should be directed to a special path as described in the previous subsection. The special path will use an alternate implementation that ensures the spurious exception is not raised.

For performance reasons the computation path must raise the required exception just once when the result underflows/overflows.

Setting the required exception flags is relatively easy. We ensure that exactly one instruction in our code sequence sets the flag we want. In many cases, the required flag is naturally set during the computation (e.g. when the result overflows/underflows). In other cases, usually special argument paths, we artificially insert an instruction that sets the required flag.

To ensure that exceptions are handled correctly when enabled, however, the specific operating system requirements need to be met. As a consequence, an additional performance penalty is incurred since more registers need to be saved upon routine entry.

7 Comparison with Existing Libraries

In this section we compare the performance and accuracy of our library (the optimized math library for Intel IA-32) against the GNU libm and CRlibm [9]. CRlibm is a recent project that attempts to return correctly rounded results for all arguments. For a fair comparison in terms of execution time, we will use the first step (quick phase) of CRlibm in our performance table below.

The accuracy of math functions in the three libraries was estimated with the Intel Math Library Test Suite (IMLTS). This tool is used extensively to validate our math routines. The largest errors found for a set of frequently used functions is given in Table 2. These errors were obtained by testing a very large number of cases, covering all execution paths.

Function	Intel libm	GNU libm	CRlibm (first step)	CRlibm
SIN	0.515082	2.60E+33	0.5	0.5
COS	0.51851	2.60E+33	0.500001	0.5
TAN	0.541852	2.60E+33	0.500001	0.5
ASIN	0.535745	3.440871	0.50157	0.5
ACOS	0.531348	2.15E+05	Absent	Absent
ATAN	0.542333	0.500307	0.5	0.5
EXP	0.540348	0.787214	0.500047	0.5
LOG	0.501397	0.500376	0.500583	0.5
POW	0.506688	8.48E+08	Absent	Absent
SINF	0.513276	1.13E+15	Absent	Absent
COSF	0.509615	3.52E+13	Absent	Absent
TANF	0.504488	7.04E+13	Absent	Absent
ASINF	0.500003	0.5	Absent	Absent
ACOSF	0.500003	0.5	Absent	Absent
ATANF	0.520918	0.5	Absent	Absent
EXPF	0.506582	0.5	Absent	Absent
LOGF	0.502916	0.5	Absent	Absent
POWF	0.501025	0.5	Absent	Absent

Table 2. Measured Accuracy of Single and Double Precision Functions

In addition to accuracy evaluation, our test suite checks whether the status flags are set correctly. As mentioned in the previous section, we allow only Inexact and Denormal to be set incorrectly (and Denormal is not mandated by the IEEE 754 standard [6]). The two libraries we compared against occasionally set some other flags incorrectly. More notably, the GNU pow() sets all status flags incorrectly. CRlibm incorrectly sets Overflow for exp() and Divide-by-Zero for log(), in both quick mode and correct rounding mode.

As an example, Table 3 shows performance numbers measured on an Intel® Pentium® 4 2.8 GHz processor running Linux. The execution times are measured in cycles. In terms of relative performance, similar results can be seen on other Intel processors.

One can see that on the Intel IA-32 architecture, our optimized library is faster than both the GNU libm and CRlibm. In most cases our library is much more accurate than the GNU libm. The GNU routines appear to be very basic implementations based on the x87 transcendental primitives. This explains their low accuracy, since the x87 primitives do not guarantee full accuracy over their entire domain and further refining is needed. As to be expected, CRlibm is very accurate even in quick mode; the very low ulp errors essentially mean that only a very small percentage of the results are rounded incorrectly. However, this comes at a cost in terms of execution time, as seen in the performance table. Our accuracy goal of 0.55 ulp means that less than 10% of returned results differ from the correctly rounded

Function	Intel libm	GNU libm	CRlibm (first step)
SIN	164	258	460
COS	164	256	455
TAN	270	304	732
ASIN	203	498	1244
ACOS	206	498	
ATAN	155	338	730
EXP	145	444	528
LOG	159	342	365
POW	249	665	
SINF	66	231	
COSF	65	232	
TANF	86	283	
ASINF	82	357	
ACOSF	84	357	
ATANF	69	302	
EXPF	48	354	
LOGF	57	197	
POWF	115	594	

Table 3. Performance of Single and Double Precision Functions [clock cycles]

result (by 1 ulp). In practice, many of our functions have maximum errors that are small enough to guarantee a much lower percentage of incorrectly rounded results.

8 Final Observations and Conclusion

Achieving fast execution times while meeting the various math function requirements for accuracy and correct behavior in special cases is a challenging task. We presented our basic design principles and some of the techniques employed to reach this goal on the Intel IA-32 architecture. Our discussion concentrated on single and double precision routines and SSE/SSE2-centered design, which typically offers the best solution for these two precision formats.

Our accuracy goal of 0.55 ulp is met by careful design which ensures that accumulated errors before the final rounding do not exceed 0.05 ulp. Sources of accumulated error include rounding errors in intermediate floating-point operations and estimation errors that are built-in by algorithm design, such as round-off errors in table values and polynomial approximation errors. Some design techniques for achieving this accuracy goal were outlined in Section 4. While most routines have not been formally verified, validation through design reviews and extensive testing gives us high confidence in the accuracy of our library.

It should be mentioned here that since modern compilers are getting better, it is often possible to obtain very good performance without going through the effort of optimization in assembly language, as long as the algorithm employs techniques well-suited to the architecture such as the ones outlined in this paper. Careful consideration of microarchitecture-specific issues and assembly-written routines remain the choice for finely tuned performance. Writing accurate math functions in a high-level language can have challenges of its own, which are not the topic of this paper. For more information, see [10].

The Intel optimized math library is made available to the public with the Intel compilers [11].

Special thanks are due to Evgeny Gvozdev of Intel Corporation, who maintains our math library test suite and provided us with the accuracy and performance data in Section 7.

References

1. International Standard ISO/IEC 9899:1999
2. *IA-32 Intel Architecture Software Developer's Manual*, volume 1: *Basic Architecture*, order number 253665, volumes 2A, 2B: *Instruction Set Reference*, order numbers 25366, 253667, Intel Corporation, January 2006.
3. *IA-32 Intel Architecture Optimization Reference Manual*, order number 248966-012, Intel Corporation, June 2005.
4. *Intel Extended Memory 64 Technology Software Developer's Guide*, order number 300834-002, Intel Corporation.
5. Cristina Iordache and Ping Tak Peter Tang, *An Overview of Floating-Point Support and Math Library on the Intel® Xscale^{TM} Architecture*, Proc. 16th IEEE Symposium on Computer Arithmetic, June 2003, pp. 122-128.
6. *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985
7. Dekker, T. J., *A Floating-Point Technique for Extending the Available Precision*, Numerische Mathematik, vol.18, pp. 224 - 242.
8. Ping Tak Peter Tang, *Table-driven Implementation of the Logarithm Function in IEEE Floating-Point Arithmetic*, ACM Transactions on Mathematical Software, vol. 16, no. 4, December 1990, pp. 378-400.
9. Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, Jean-Michel Muller, *CR-LIBM A Library of Correctly Rounded Elementary Functions in Double-Precision*, `http://lipforge.ens-lyon.fr/projects/crlibm`, April 2006.
10. Eric Fleegal, *Microsoft Visual C++ Floating-Point Optimization*, June 2004, `http://msdn.microsoft.com/library/`.
11. Intel Software Development Products, `http://www.intel.com/cd/software/products/asmo-na/eng/index.htm`

The Sunity Representation to Improve the Accuracy of Some Computations*

Tomás Lang[1] and Javier D. Bruguera[2]

[1] Dept. Electrical Engineering and Computer Science.
University of California at Irvine.
Irvine. CA 92627. USA
tlang@uci.edu

[2] Dept. Electronic and Computer Engineering.
University of Santiago de Compostela.
15706 Santiago de Compostela. SPAIN.
bruguera@dec.usc.es

Abstract. A modification of the floating-point representation is presented that reduces the relative representation error around the value 1.0. The objective of this reduction is to avoid the massive increase in relative error in some important computations, such as $1 - f(x)$, $\arccos(x)$, $\ln(1 + x)$ and $(1 + x)^n$.
The idea is to implement the basic operations for this representation in hardware so that it would not be necessary to provide software solutions for this loss of accuracy. The paper shows classes of computations that benefit from the proposed representation and gives expressions for the corresponding relative errors.

1 Introduction

Floating-point representation is characterized by a large dynamic range and a uniform relative representation error over the whole range of normalized values. This provides the basis for high accuracy computations. Nevertheless, a loss of accuracy occurs in some computations [1], among others a typical example being cancellations, where the relative error is multiplied by the cancellation factor.

The resulting loss of accuracy can be mitigated by an increase in the precision of the representation, but this has the disadvantage of higher storage requirements and more complex and slower operations. Moreover, in some cases even doubling or quadrupling the precision might not be sufficient. Changes in the algorithm to avoid the problem are also possible in some cases.

In this paper we propose a modification of the floating-point representation to handle some of the above-mentioned issues. The main idea is to reduce the relative representation error around the value 1. This representation is intended to be used by the hardware implementation of the operations and by the library functions so as to improve the accuracy for some important computations [5, 6], such as $1-f(x)$, with $f(x)$ close to 1, $\arccos(x)$ for x close to 1, and $\ln(1+x)$ and $(1 + x)^n$ for x close to 0. Although this certainly does not address most of the accuracy problems, it deals with some important computations and can serve as preliminary ideas for more general approaches. In this vane, we show some cases that can be easily transformed to be handled by the proposed representation.

The loss of accuracy that occurs in the computations of the types mentioned above is well known and have been addressed by expert numerical analysts and programmers by changing

* Work supported in part by Ministry of Education and Science of Spain and UE (FEDER funds), and Office for R&D of Galicia (Spain) under contracts TIN2004-07797-C02 and PGIDT05PXI20602PN.

the algorithms and/or by implementing appropriate library functions [5]. However, these approaches are quite specific and require a particular awareness and analysis for each case. The approach taken here is to provide representation and hardware support that is applicable to a wider class of computations and can be used automatically by the system without additional intervention. A more selective utilization is also possible, in which programmers have some control of the use of the representation. It is certainly still unclear whether this approach is beneficial for more complex computations, where the accumulation of errors might make the reduced error of the representation ineffective.

The proposed representation is intended to be used in general-purpose processors as well as in application specific ones, examples of the latter type being processors for graphics and DSP.

In [3] we presented the unity representation, which uses the whole representation space for the range [-1,1] and has a reduced representation error close to the values -1 and 1. We showed high accuracy results for some of the above mentioned computations. We also showed the advantage of the unity representation over a fixed-point representation, which might be considered ideal for this limited range. Finally, we presented a way of converting a cancellation to a form that can use the unity representation and applied this to compute $\pi/2 - \arccos(x)$ for small x.

Using the unity representation requires special instructions to operate on values in this representation. Moreover, it can not be easily used for computations in which some values are in other ranges. It cannot be used either with values having a larger range, but with accuracy issues close to 1 (for example $z = 1 - x$ for $x \leq 20$).

In this paper we present the sunity representation, which extends the unity representation to include also values larger than 1. This allows its use for computations such as $e^x - 1$, $(1+x)^n$, and $\ln(1+x)$, with small x. Moreover, we suggest a way to include the sunity representation into the floating-point representation, which would eliminate the limitations discussed above for the unity representation.

We show several classes of computations that benefit from the proposed representation and present expressions for the relative error, both using the standard floating-point representation as well as the proposed representation.

As an extension to this work, we are developing the algorithms for the implementation of the basic operations. Moreover, we are considering the effect of the proposed representation in more complex algorithms, examples being computations required for 3D graphics, such as the computation of the rotation angle produced by a composition of two 3D rotations.

For compactness, in the numerical examples of the paper we use 32–bit representations. The extension to other precisions is straightforward and produces similar results. All these representations include a sign bit, but for the sake of simplicity, we restrict only to positive values. Moreover, we do not consider denormals.

2 Unity representation for unit range numbers

In this section we summarize the main characteristics of the unity representation and compare it with the standard floating–point representation. Note that the unity representation is only valid for the representation of numbers in the range $[-1, 1]$. For a detailed description of this representation and examples of computations involving unit–range numbers, see [3].

Consider numbers in the range $[0, 1)$. The characteristics of the floating-point (FP) and fixed-point (FX) representations are summarized in Table 1. For this range, the FP repre-

	FP	FX	Unity
Min. value	2^{-127}	2^{-31}	2^{-127}
Max. value	$1-2^{-23}$	$1-2^{-31}$	$1-2^{-127}$
Diff. close 0	2^{-150}	2^{-31}	2^{-150}
Diff. close 0.5	2^{-24}	2^{-31}	2^{-25}
Diff. close 1	2^{-24}	2^{-31}	2^{-150}

Table 1. Characteristics of the 32–bits representations in range $[0, 1)$.

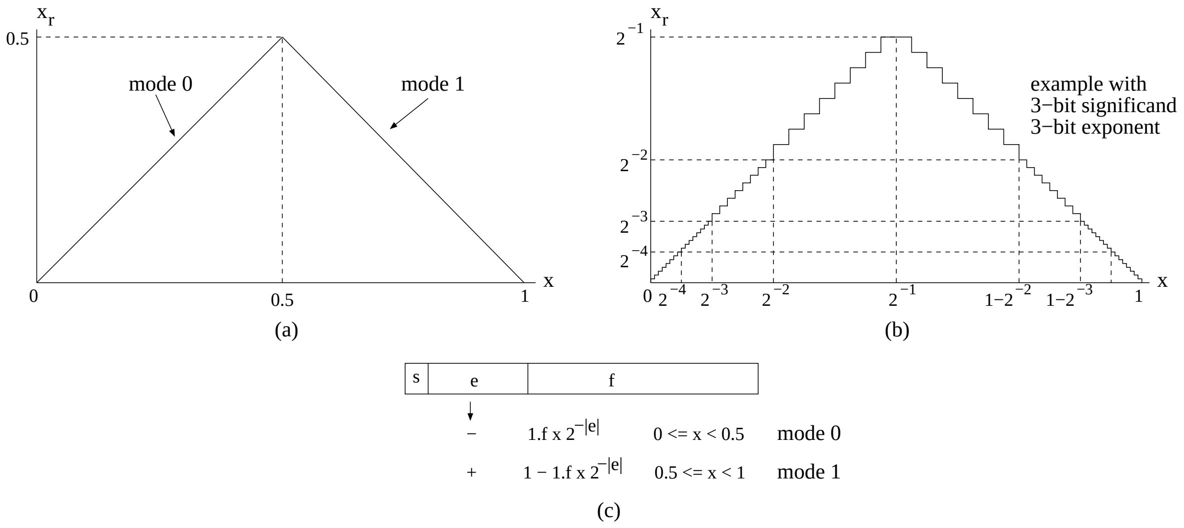


Fig. 1. Unity representation. (a) General representation. (b) Example with 3–bit significand and 3–bit exponent. (c) Specification of the two modes.

sentation [2] uses only negative exponents, so that half of the combinations are not used. Moreover, the density close to 1 is much smaller than the density close to zero, and the density varies from binade to binade. This low density close to 1 produces inaccurate results in some operations. See [3] for examples of inaccurate results in the computation of the arcccos function with small angles, exponential function with negative exponents and computation of a rotation angle. For the FX representation the density close to 1 is larger than in the FP representation, but it still produces inaccurate results in the examples above. Moreover, it is not able to represent very small values.

To solve the issues discussed for floating-point and fixed-point representations for values in the range $[0, 1)$, the **unity representation** was recently proposed. This representation is symmetrical with respect to 0.5 such that for x a real number in the range $[0, 1)$, the unity representation x_r is

$$x_r = \begin{cases} \mathrm{round}(x) & \text{if } 0 \leq x < 0.5 \text{ (mode 0)} \\ \mathrm{round}(1-x) & \text{if } 0.5 \leq x < 1 \text{ (mode 1)} \end{cases} \tag{1}$$

where x_r is a FP number and $\mathrm{round}(x)$ corresponds to the rounding to nearest rounding mode. This representation is shown in Figures 1(a) and (b). The density is the same close to 1 and close to 0. That means that now we can represent values as close to 1 as to 0. Moreover, we keep the property of the floating-point representation that allows much smaller values than for fixed-point representation, as shown in Table 1. To specify the two modes, we make use

	Standard FP	
	representation	numerical value
θ	$1.00010000000000100001000 \times 2^{-15}$	$1.00010000000000100001000 \times 2^{-15}$
$\cos(\theta)$	$1.00000000000000000000000 \times 2^{0}$	$1.00000000000000000000000 \times 2^{0}$
$\arccos(\cos(\theta))$	$0.00000000000000000000000 \times 2^{-127}$	$0.00000000000000000000000 \times 2^{-127}$

	FX	
	representation	numerical value
θ	.00000000000000100010000000000010	.00000000000000100010000000000010
$\cos(\theta)$	.11111111111111111111111111111110	.11111111111111111111111111111110
$\arccos(\cos(\theta))$	.00000000000000101101010000001010	.00000000000000101101010000001010

	Unity	
	representation	numerical value
θ	$1.00010000000000100001000 \times 2^{-15}$	$1.00010000000000100001000 \times 2^{-15}$
$\cos(\theta)$	$1.00100001000001000110001 \times 2^{31}$ (*)	$1 - 1.00100001000001000110001 \times 2^{-31}$
$\arccos(\cos(\theta))$	$1.00010000000000100001000 \times 2^{-15}$	$1.00010000000000100001000 \times 2^{-15}$

Table 2. Arccosine with different representations. (*) Positive exponent to represent $1 - x$

.

of the fact that for a value in the range $[0, 1)$, in the floating-point representation the positive exponents are not used. Then, as shown in Figure 1(c), the negative exponents are used for representation in mode 0 and the positive exponents for the representation in mode 1.

To show the advantages of the unity representation, several examples have been simulated. The three 32–bit representations, FP, FX and unity, have been simulated with Maple[7]. In all the examples, the result obtained with the unity representation is the same as the one obtained with Maple using an extended precision (40 decimal digits) rounded to the FP single–precision format, and this result is much more accurate than the result obtained with FP and FX representations.

As an example, Table 2 shows the computation of the arc cosine function for an angle $\theta = 2^{-15} + 2^{-19} + 2^{-30} + 2^{-35}$. We see that for standard floating-point representation, $\cos(\theta) = 1.0$, resulting in $\arccos(\cos(\theta)) = 0$. The FX representation produces an inaccurate result, whereas the unity representation produces the correct angle.

3 Symmetric unity (sunity) representation

The goal of the unity representation is to improve the characteristics of the FP representation for numbers close to, but smaller than, 1. However, there are a number of computations that require to compute values around 1. Examples are

1. Cancellations $1 - f(x)$ or $f(x) - 1$, with $f(x) \to 1$. Among these we can cite $1 - \cos(x)$, $e^x - 1$ and $1 - (2/\pi)\arccos(x)$ for small x.
2. Functions $f(x)$ with x close to 1. For example, $\arccos(x)$, $(1 + y)^n$ being y small, and $\ln(1 + y)$ being y small.

In this section we extend the underlying idea of the unity representation to propose another representation, the *symmetric unity representation (sunity)*, which is applicable to numbers in the range $[-2, 2]$.

In the unity representation, numbers $x \in [0.5, 1)$ are represented as $x_r = \text{round}(1 - x)$ (see equation (1)). That is, these numbers are represented by means of a small *negative*

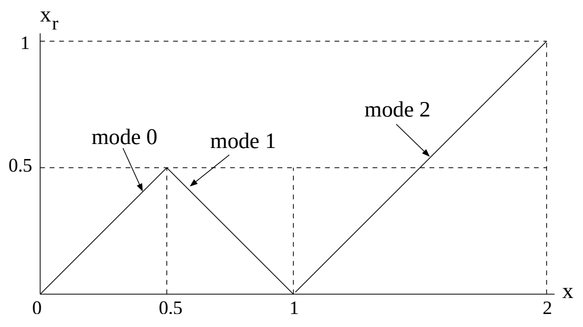


Fig. 2. Symmetric Unity representation.

displacement from 1, in such a way that the closer the number to 1 the smaller the absolute value of the displacement representing it; the maximum displacement is 0.5, used to represent the real number $x = 0.5$. This is shown in figure 1(a).

This same idea is used for the symmetric unity representation. In this case, we use a *positive displacement* to represent numbers larger than 1. This way, the symmetric unity representation of a real number x in the range $[0, 2)$ is

$$x_r = \begin{cases} \text{round}(x) & \text{if } 0 \leq x < 0.5 \text{ (mode 0)} \\ \text{round}(1 - x) & \text{if } 0.5 \leq x < 1 \text{ (mode 1)} \\ \text{round}(x - 1) & \text{if } 1 \leq x < 2 \quad \text{(mode 2)} \end{cases} \tag{2}$$

This representation is shown in Figure 2. Note that the only difference with respect to the unity representation is that we have included the representation of real numbers between 1 and 2. The representation of these numbers is a floating–point number smaller than 1, which represents the displacement from 1, in such a way that, the closer the number to 1 the smaller is the displacement.

Figures 3 and 4 show the absolute and relative errors of the FX, FP and symmetric unity representations for real numbers $x \in [0, 2)$. The Figures show the maximum error in each binade[3]. The absolute and relative errors are calculated as

$$\begin{aligned} \text{abs err}(x) &= |x - \text{repres}(x)| \\ \text{rel err}(x) &= |x - \text{repres}(x)|/|x| \end{aligned} \tag{3}$$

repres(x) being the FX or FP representations of the real number x and, in case of the symmetric unity representation, the numerical value of the symmetric unity representation of x. Table 3 shows the values of these maximum errors. From the figures we see that the sunity representation produces a significant reduction of the error around the value 1. More precisely, for $x = 1 + a \times 2^{-d}$, with $1 \leq |a| < 2$, the relative errors are

- For floating-point representation $r_x \leq \min(0.5 \times 2^{-u}, a \times 2^{-d})$
- For sunity representation $r_x \leq 2^{-(d+u)}$

where 2^{-u} is a unit in the last position (*ulp*).

The reduction in relative error close to 1.0 provided by the sunity representation is used to avoid the loss of accuracy of some computations, as described in Section 5.

[3] We are considering that, for the symmetric unity representation, there are several binades between 0.5 and 1 and between 1 and 2 as well, defined by the symmetric unity representation x_r of a given real number $0.5 \leq x < 2$. For the FX representation we have considered the same binades as for the FP representation.

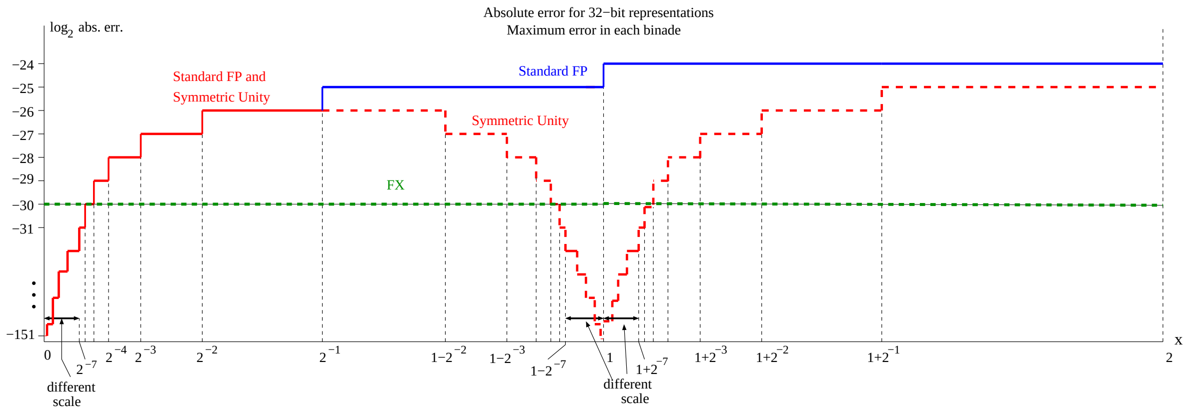


Fig. 3. Absolute errors in interval $[0, 2)$ of the 32–bit FX, FP and symmetric unity representations.

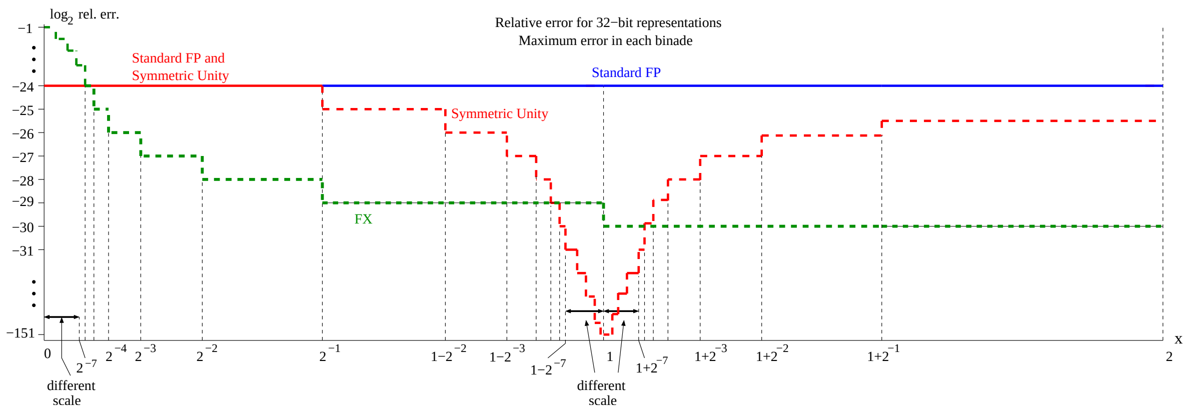


Fig. 4. Relative errors in interval $[0, 2)$ of the 32–bit FX, FP and symmetric unity representations.

Repres.	absolute error	relative error	
FX	2^{-30}	2^{i-30}	
FP	$2^{-24} \times 2^{exp\ fp}$	2^{-24}	
Sunity	$2^{-24} \times 2^{exp\ sunity}$	2^{-24}	$(x < 0.5)$
		$(2^{-23} \times 2^{exp\ sunity})$	$(0.5 \leq x < 1)$
		$(2^{-24} \times 2^{exp\ sunity})$	$(1 \leq x < 2)$

Table 3. Maximum value per binade of the absolute and relative errors for the 32–bit representations. *exp fp* is the exponent of the FP representation, i is the number of leading zeros in the FX representation, and *exp sunity* is the exponent of the sunity representation x_r (see equation(2)).

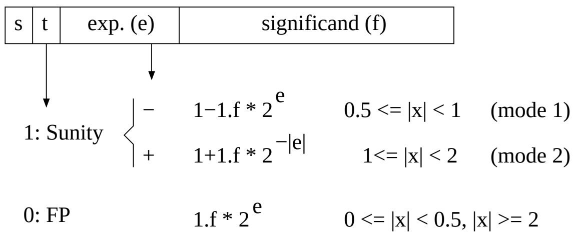

s
t
exp. (e)
significand (f)
1: Sunity
− 1−1.f * 2^e 0.5 <= |x| < 1 (mode 1)
+ 1+1.f * 2^−|e| 1<= |x| < 2 (mode 2)
0: FP 1.f * 2^e 0 <= |x| < 0.5, |x| >= 2

Fig. 5. Fields in the sunity representation.

4 Combining the sunity and FP representations

The unity representation was presented in [3] as an operand type different from the standard floating–point operand. That is, to use these representations the programmer should specify which variables are unity and use them properly with special instructions, only in those computations with limited range for the inputs and results. This model, although it improves the accuracy with respect to the FP representation, is restrictive from the programmer point of view. For the sunity representation, we propose a different model: a combined representation FP plus sunity with improved accuracy in the range $[0.5, 2)$. No special instructions are required to manage the representation, but the standard instruction set should determine the range of operands and results. In this way, it is possible to use operands of different type, sunity or FP, in the same instruction in a transparent way to the programmer.

Specifically, the representation consists of two types: the FP type, for values less than 0.5 and greater or equal to 2, and the sunity type, for the range [0.5,2). Moreover, the sunity type is divided into two modes: mode 1 corresponding to the binade [0.5,1) and mode 2 corresponding to the binade [1,2). A possible format for this combined representation is shown in Figure 5. As can be seen, bit t is used to differentiate between the two types. To differentiate among the two modes of the sunity type we make use of the fact that the exponent for this type is not positive. Consequently, we can use the sign of the exponent to indicate the mode. Since one bit is used to indicate the type, this representation looses one bit, with respect to the standard floating-point representation, this bit can be taken from the exponent (reducing the dynamic range) or from the significand (reducing the precision).

5 Types of computations that benefit from the sunity representation

We now describe the types of computations that might benefit from the sunity representation. In general, these correspond to computations for which the relative error of the result is larger than the relative error of the argument. This corresponds to a reduction of accuracy for FP

Type of computation	Computation	Relative error FP	Relative error Sunity
	x	$\min(0.5 \times 2^{-u}, a \times 2^{-d})$	$2^{-(d+u)}$
	y	0.5×2^{-u}	0.5×2^{-u}
Type 1	$1-x$	$\min(0.5 \times 2^{-(u-d)}, 1)$	0.5×2^{-u}
	$1-\cos(y)$	$\min(0.5 \times 2^{-(u-2d-1)}, 1)$	0.5×2^{-u}
	$e^y - 1$	$\min(0.5 \times 2^{-(u-d)}, 1)$	0.5×2^{-u}
Type 2	$\pi/2 - \arccos(y)$	$2^{-(u-d-1)}$	2^{-u}
Type 3	$\arccos(x)$	$\min(0.25 \times 2^{-(u-d)}, 1)$	0.5×2^{-u}
	$\ln(x)$	$\min(0.5 \times 2^{-(u-d)}, 1)$	0.5×2^{-u}
Type 4	$(1+y)^n$	$0.5 \times 2^{-u} +$ $n \times \min(0.5 \times 2^{-u}, y)$	$0.5 \times 2^{-u} +$ $n \times 0.5 \times 2^{-(u+d)}$
Type 5	$(b+y)^n$, b exact	$0.5 \times 2^{-u} + (n/b) \times$ $(0.5 \times 2^{-u} + \min(0.5 \times 2^{-u}, y))$	$2^{-u} + n \times 0.5 \times 2^{-(u+d)}$

Table 4. Relative error of several operations with argument close to 1 ($x = 1 + a \times 2^{-d}$) or close to 0 ($y = a \times 2^{-d}$), for $1 \leq |a| < 2$ and $d \geq 1$. We denote by 2^{-u} the unit in the last position (ulp). Rounding is to nearest.

representation, which is eliminated by using the sunity representation. Example computations are shown in Table 4 together with approximate expressions for the relative error. Note that some of the functions in the table have arguments close to 1 and other functions have arguments close to 0. We denote as x the arguments close to 1 and y the arguments close to 0. Derivations of these errors are given in [4]. We have identified the following types:

Type 1: Cancellations of the type $z = 1 - f(w)$, with $f(w)$ close to 1.
Examples of this type are $x - 1$, $1 - \cos(y)$, and $e^y - 1$.

Type 2: Cancellations of the type $z = a - b$, with b close to a.
This is a generalization of Type 1. To perform this computation using the sunity representation we transform it to

$$z = a - b = b \times (a/b - 1) = b \times (q - 1) \tag{4}$$

This eliminates the increase in relative error if q is computed with high precision using the sunity representation. Note that this improvement is not obtained, for instance, if the division is performed with a and b in floating–point representation, since the division would produce a maximum relative error of 2^{-u}.
An example of this type is $z = \pi/2 - \arccos(y)$ for small y. In this case, to achieve the desired accuracy the function $(2/\pi)\arccos(y)$ has to be computed with high precision when y is close to 0 and its value represented in sunity. In this particular case, the computation can be done in floating point by defining the library function $\pi/2 - \arccos(y)$.

Type 3: Reduction in value without cancellation.
Type 1 and Type 2 above are characterized by preserving (roughly) the absolute error while increasing the relative error. This increase is due to the reduction in value between the operand and the result. A similar phenomenon can occur for computations in which this reduction in value is produced, without cancellation.
Examples of this type are $\arccos(x)$ and $\ln(x)$ with x close to 1 and the computation of the angle of a 3D rotation [3].

Type 4: Increase of error due to large derivative.

	Standard FP repres.	Sunity repres.
y	$1.00000000011000000000000 \times 2^{-19}$	
e^y	$1.00000000000000000010000 \times 2^{0}$	$1.00000000011000000001000 \times 2^{19}$ (*)
$e^y - 1$	$1.00000000000000000000000 \times 2^{-19}$	$1.00000000011000000001000 \times 2^{-19}$

	Standard FP repres.	Sunity repres.
Operand (x)	$1 + 2^{-19} + 2^{-30} + 2^{-37}$	
x	$1.00000000000000000010000 \times 2^{0}$	$1.00000000001000001000000 \times 2^{19}$ (*)
$\ln(x)$	$1.11111111111111111110000 \times 2^{-20}$	$1.00000000001000000011000 \times 2^{-19}$

Table 5. Some examples of computations using standard FP and sunity 32–bit representations. Note that in these examples, the sunity representation obtains a value close to 0 with larger accuracy than the FP representation. ($*$) Positive exponent to represent $1 + \mathit{value}$.

For computations in which the derivative is large for argument value close to 1, the absolute error is increased for that value of the argument. By using the sunity representation it is possible to avoid a loss of accuracy

Example of this type is $z = (1 + y)^n$ for y small.

Type 5: Generalization of Type 4. The Type 4 case can be generalized to

$$z = (b + y)^n$$

with y/b small and b exact. To use the sunity representation we convert the computation as follows:

$$z = (b + y)^n = b(1 + y/b)^n$$

and represent $1 + y/b$ in sunity. As indicated in [4], the desired accuracy is obtained only when b is exact.

Some examples of the above computations are summarized in table 5. The table shows the results obtained using the single-precision FP and the 32-bits sunity representations. As it can be seen, the sunity representation produces more accurate results. We have used Maple to simulate both representations. In all the examples, the result obtained with the sunity representation is the same as the one obtained with Maple using an extended precision (40 decimal digits) rounded to the FP single–precision format. Although we use in our examples single-precision representation (32 bits), the utilization of a larger wordlength for each representation, for example 64–bits, will result in the same conclusion: The sunity representation is more accurate than the floating–point representation.

6 Conclusions

We have presented a floating-point representation that has a smaller relative representation error close to the value 1 than the standard FP representation. This representation has been shown to be effective in some cases in which the standard representation results in a reduction of accuracy. We have classified the situations where this occurs and have illustrated some specific examples by giving expressions for the relative error. Although this proposal focuses on cases in which the accuracy issue is related to values close to 1, we have shown some situations in which a computation can be transformed to make use of this representation. This might lead to more general developments. In this vane, we are considering the use of this representation in more complex computations in which, in addition to the representation errors, accumulated errors have to be included.

The proposed unified representation requires one bit to differentiate between the FP and the sunity portion. This bit can be obtained from the exponent, reducing in this way the range, or from the significand, with a reduction of one bit of precision. Which of these solutions is best might depend on the overall number of bits (for instance single or double precision) and on the application. Except for the loss of this bit, there seem to be no situations in which there is a loss of accuracy because of the use of the proposed representation.

The use of the representation requires a modification in the implementation of the operations and of the library functions. These modifications will introduce some added complexity; this should be done in a way that does not significantly affect the performance of the operations for the FP mode. Moreover, the added complexity to use the operands (arguments) and/or produce results in the sunity mode might not be beneficial in some cases. To avoid the corresponding reduction in performance, it might be convenient to provide some way to disable this mode, with this capability being used by sophisticated programmers.

A more thorough analysis of the characteristics of the overall representation would be desirable. In particular it would be necessary to analyze the effect of the transitions between the FP mode and the sunity mode.

References

1. N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* (Chapter 1). Siam. (1996).
2. *IEEE Standard for binary floating–point arithmetic, ANSI/IEEE Std. 754-1985.* The Institute of Electrical and Electronic Engineers, Inc., New York. (1985).
3. T. Lang and J. D. Bruguera. *Representation of Unit–Range Numbers.* Proc. 39th Asilomar Conference on Signals, Systems and Computers. (2005)
4. T. Lang and J. D. Bruguera. *Derivation of the relative errors for the sunity representation.* Appendix. (available at `http://www.ac.usc.es`). (2006).
5. P. Markstein. *IA-64 and Elementary functions.* Ed. Prentice Hall. (2000).
6. J-M. Muller. *Elementary Functions. Algorithms and Implementations.* Ed. Birkhäuser. (1997).
7. Waterloo Maple Inc. *Maple 9.01.* (2003)

RN-Codings: New Insights and Some Applications

Peter Kornerup[1] and Jean-Michel Muller[2]

[1] University of Southern Denmark
Odense, Denmark
kornerup@imada.sdu.dk
[2] CNRS-LIP-Arénaire
Lyon, France
Jean-Michel.Muller@ens-lyon.fr

Abstract. During any composite computation there is a constant need for rounding intermediate results before they can participate in further processing. Recently a class of number representations denoted RN-Codings were introduced, allowing rounding to take place by a simple truncation, with the additional property that problems with double-roundings are avoided. In this paper we investigate a particular encoding of the binary representation, and conversions between the RN-Coding in this encoding and ordinary 2's complement representation. This encoding is essentially an ordinary 2's complement with an appended round-bit, but still allowing double-rounding without errors. Conversions from 2's complement to RN-Coding can be performed in constant time, whereas conversion the other way in general takes at least logarithmic time. A very fast parallel prefix algorithm for this conversion is defined and analyzed. Based on the observation that the signs of the non-zero digits of an RN-Coded number alternate, a sketch of how forcing such alternation in the coefficients of CORDIC expansions could be exploited.

1 Introduction

In a recent paper [KM05] a class of number representations denoted RN-Codings were introduced, the "RN" standing for "round to nearest", as these radix-β, signed-digit representations have the property that truncation yields rounding to the nearest representable value. They are based on a generalization of the observation that certain radix representations are known to posses this property, e.g., the balanced ternary ($\beta = 3$) system over the digit set $\{-1, 0, 1\}$ is an example. Another such representation is obtained by performing the original Booth-recoding [Boo51] on a 2's complement number into the digit set $\{-1, 0, 1\}$, where it is well-known that the non-zero digits of the recoded number alternate in sign.

We shall in Section 2 briefly from [KM05] cite some of the definitions and properties of the general RN-Codings/representations. However, we will here explore the binary representation, e.g., as obtained by the Booth recoding, the rounding by truncation property, including the feature that the effect of one rounding followed by another rounding yields the same result, as would be obtained by a single rounding to the same accuracy as the two combined.

Section 3 then analyzes conversions between RN-Codings and 2's complement representations. Conversion from the latter to the former is performed by the Booth algorithm, yielding a signed-digit/borrow-save representation in a straightforward encoding, which for an n-digit word requires $2n$ bits. It is then realized that $n + 1$ bits are sufficient, providing a simpler alternative encoding consisting of the bits of the truncated 2's complement encoding, with a round-bit appended.

Conversion the other way, from RN-Coding in this "canonical" encoding into 2's complement representation (essentially adding in the round-bit) is then shown to be realizable by a parallel prefix structure, which is then analyzed. Section 3 contains examples on some composite computations where fast and optimal roundings are useful, and may come for free when RN-coding is employed. Although not based on these representations, the idea of forcing alternating signs in expansions is applied to a CORDIC algorithm [Vol59] as a sketch to be pursued. Section 4 then concludes with some examples of applications.

2 Definitions and Basic Properties (cited from [KM05])

Definition 1 (RN-codings). *Let β be an integer greater than or equal to 2. The digit sequence $D = d_n d_{n-1} d_{n-2} \cdots$ (with $-\beta + 1 \le d_i \le \beta - 1$) is an RN-coding in radix β of x iff*

1. *$x = \sum_{i=-\infty}^{n} d_i \beta^i$ (that is D is a radix-β representation of x);*
2. *for any $j \le n$,*
$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \le \frac{1}{2} \beta^j,$$
that is, if the digit sequence is truncated to the right at any position j, the obtained sequence is always the number of the form $d_n d_{n-1} d_{n-2} d_{n-3} \ldots d_j$ that is closest to x.

Hence, truncating the RN-coding of a number at any position is equivalent to rounding it to the nearest.

Although it is possible to deal with infinite representations, we shall here restrict our discussions to finite representations. The following observations on such RN-Codings for general $\beta \ge 2$ are then easily found:

Observation 2 (Characterizations of finite RN-codings).

– *if $\beta \ge 3$ is odd, then $D = d_m d_{m-1} \cdots d_\ell$ is an RN-coding iff*
$$\forall i, \frac{-\beta + 1}{2} \le d_i \le \frac{\beta - 1}{2};$$
– *if $\beta \ge 2$ is even then $D = d_m d_{m-1} \cdots d_\ell$ is an RN-coding iff*
  1. *all digits have absolute value less than or equal to $\beta/2$;*
  2. *if $|d_i| = \beta/2$, then the first non-zero digit that follows on the right has an opposite sign, that is, the largest $j < i$ such that $d_j \ne 0$ satisfies $d_i \times d_j < 0$.*

Observe that for odd β the system is non-redundant, whereas for β even the system is redundant, in the sense that non-zero numbers have two representations. In particular note that for radix 2 the digit set is $\{-1, 0, 1\}$, known by the names of "signed-digit" or "borrow-save". Also, since here the non-zero digits all have absolute value one, their signs alternate. From now on we will only deal with radix 2 representations and algorithms for these.

An important property of the RN-coding is that no errors are introduced if repeated roundings take place, thus avoiding the *double rounding* problem with some roundings. It may happen when the result of first rounding to a position j, followed by rounding to position k, does not yield the same result as if directly rounding to position k. We repeat from [KM05] the following obvious result:

Observation 3 (Double rounding).
Let $\mathrm{rn}_i(x)$ be the function that rounds the value of x to nearest at position i. Then for $k > j$, if x is represented in the RN-Coding, then
$$\mathrm{rn}_k(x) = \mathrm{rn}_k(\mathrm{rn}_j(x))$$

3 Converting between RN-Coding and 2's Complement

3.1 Conversion from 2's Complement to RN-Coding

Consider an input value $x = -b_m 2^m + \sum_{i=\ell}^{m-1} b_i 2^i$ in 2's complement representation:

$$x \sim b_m b_{m-1} \cdots b_{\ell+1} b_\ell$$

with $b_i \in \{0,1\}$ and $m > \ell$. Then the digit string

$$\delta_m \delta_{m-1} \cdots \delta_{\ell+1} \delta_\ell \quad \text{with} \quad \delta_i \in \{-1, 0, 1\}$$

defined (by the Booth recoding [Boo51]) for $i = \ell, \cdots, m$ as

$$\delta_i = b_{i-1} - b_i \quad (\text{with } b_{\ell-1} = 0 \text{ by convention}) \tag{1}$$

is an RN-Coding of x with $\delta_i \in \{-1, 0, 1\}$. That it represents the same value follows trivially by observing that the converted string represents the value $2x - x$. The alternation of the signs of non-zero digits is easily seen by considering how strings of the form $01 \cdots 10$ and $10 \cdots 01$ are converted.

Thus the conversion can be performed in constant time. Actually, the digits of the 2's complement representation directly provides for an encoding of the converted digits as a tuple: $\delta_i \sim (b_{i-1}, b_i)$ for $i = \ell, \cdots, m$ where

$$\begin{aligned} -1 &\sim (0,1) \\ 0 &\sim (0,0) \text{ or } (1,1) \\ 1 &\sim (1,0), \end{aligned}$$

where the value of the digit is the difference between the first and the second component.

Example 4. Let $x = 110100110010$ be a sign-extended 2's complement number and write the digits of $2x$ above the digits of x:

$2x$	1	0	1	0	0	1	1	0	0	1	0	0
x	1	1	0	1	0	0	1	1	0	0	1	0
x in RN-Coding		$\bar{1}$	1	$\bar{1}$	0	1	0	$\bar{1}$	0	1	$\bar{1}$	0

where it is seen that in any column the two upper-most bits provide the encoding defined above of the signed-digit below in the column. Since the digit in position $m+1$ will always be 0, there is no need to include the most significant position otherwise found in the two top rows.

If x is non-zero and b_k is the last non-zero digit of x from the left, then $\delta_k = -1$, confirmed in the example, hence the last non-zero digit is always $\bar{1}$ and thus unique. However, if an RN-Coded number is truncated for rounding somewhere, the resulting representation may have its last non-zero digit of value 1. But in this case the immediately preceding digit is either 0 or $\bar{1}$, and in both cases it is possible to rewrite these two digits, i.e., $01 \rightarrow 1\bar{1}$ or $\bar{1}1 \rightarrow 0\bar{1}$, such that the last non-zero digit is $\bar{1}$.

Hence there are exactly two finite binary RN-Codings of any non-zero binary number of the form $a2^k$ for integral a and k, but requiring a specific sign of the last non-zero digit makes the representation unique. On the other hand without this requirement, rounding by truncation makes the rounding unbiased in the tie-situation, by randomly rounding up or down, depending on the sign of the last non-zero digit in the remaining digit string.

Example 5. Rounding the value of x in Example 1 by truncating off the two least significant digits we obtain

$\text{rn}_2(2x)$	1	0	1	0	0	1	1	0	0	**1**
$\text{rn}_2(x)$	1	1	0	1	0	0	1	1	0	0
$\text{rn}_2(x)$ in RN-Coding		$\bar{1}$	1	$\bar{1}$	0	1	0	$\bar{1}$	0	1

where it is noted that the bit of value 1 in the upper rightmost corner (in boldface) acts as a round bit, assuring a round-up in cases there is a tie-situation as here.

The example shows that there is another very compact encoding of RN-Coded numbers derived directly from the 2's complement representation, noting in the example that the upper row need not be part of the encoding, except for the round-bit. We will denote it the *canonical encoding*, and note that it is a kind of "carry-save" in the sense that it contains a bit not yet added in. The same idea have previously been pursued in [NMLE00] in a floating-point setting, denoted "packet-forwarding".

Definition 6 (Canonical encoding).
Let the number x be given in 2's complement representation as the bit string $b_m \cdots b_{\ell+1} b_\ell$, such that $x = -b_m 2^m + \sum_{i=\ell}^{m-1} b_i 2^i$. Then the canonical encoding *of the RN-Coded representation of x is defined as the pair*

$$x \sim (b_m b_{m-1} \cdots b_{\ell+1} b_\ell, r) \quad \text{where the round-bit is } r = 0$$

and after truncation at position k, for $m \geq k > \ell$

$$\text{rn}_k(x) \sim (b_m b_{m-1} \cdots b_{k+1} b_k, r) \quad \text{with round-bit } \ r = b_{k-1}.$$

Note that this encoding is amenable to a straightforward way of performing arithmetic processing, since the round-bit may very often be taken into account at no additional cost, simply by using it as a carry-in to an adder together with the 2's complement component. The signed-digit interpretation is available in the encoding by pairing bits, (b_{i-1}, b_i) for $i > k$ and (r, b_k), when truncated at position k. Obviously, the encoding then allows another rounding by truncation.

There are other equally compact encodings of RN-Coded numbers, e.g., one could encode the signed-digit string simply by the string of bits obtained as the absolute values of the digits, together with say the sign of the most (or least) non-zero digit. Due to the alternating signs of the non-zero digits, this is sufficient to reconstruct the actual digit values. However, this encoding does not seem very convenient for arithmetic processing, as the correct signs will then have to be distributed over the bit string.

3.2 Conversion from RN-Coding to 2's Complement

The example of converting $0000000\bar{1}$ into its 2's complement equivalent 11111111 shows that it is not possible to perform this conversion in constant time, information may have to travel an arbitrary distance to the left. Hence a conversion may in general take at least logarithmic time. Since the RN-Coding is a special case of the (redundant) signed-digit representation, obviously this conversion is fundamentally equivalent to an addition.

If an RN-Coded number is in canonical encoding, conversion into 2's complement may require a non-zero round-bit to be added in, it simply consists in an incrementation for which efficient methods exists. But to complete the picture let us develop such a parallel prefix type algorithm for adding in such a round-bit. Let the operand in canonical encoding be

$$x \sim (x_m, x_{m-1} \cdots x_\ell, r),$$

then the propagate- and generate-bits for input to the trees are

$$p_i = x_i \text{ and } g_i = 0 \quad \text{for} \quad i = \ell, \ell+1, \cdots, m.$$

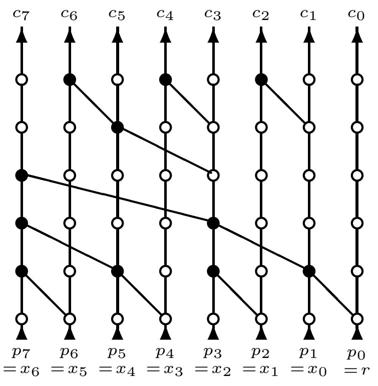
By the composition rule $(g,p)\circ(g',p') = (g+pg', pp') = (0, pp')$, so all the generate-bits will be zero, and need not be calculated. Thus the nodes of the parallel prefix tree for calculating the carries will consist of AND-operators, with input at the leaves $p_i = x_i$. Let the output of the i^{th} tree then be the combined propagate-bits

$$P_i = \bigwedge_{k=\ell}^{i} p_k \quad \text{for} \quad i = \ell, \ell+1, \cdots, m.$$

The actual carries can then be found as $c_\ell = r$ and $c_i = r\,P_{i-1}$ for $i > \ell$. However, this will require a broadcast of the round-bit to all positions. This can be avoided by changing the input p_ℓ to $p_\ell = r$ and shifting the rest over so that they are defined as $p_i = x_{i-1}$ for $i > \ell$. The final output of the conversion is then found as

$$e_i = x_i \oplus c_i \quad \text{for} \quad i = \ell, \ell+1, \cdots, m.$$

The figure below shows such parallel prefix trees following the structure suggested by Brent and Kung in [BK82] for calculating the carries of an 8 digit conversion. With $p_i = x_{i-1}$ for $i > 0$ and $p_0 = r$ as input at the bottom, and an AND operation in each of the black nodes, the carries appears at the top, to be XOR'ed with the input bits x_i.



Comparing the complexity of these parallel prefix trees with those of carry look-ahead trees for normal adders, it is found that the lengths of the paths in count of nodes through the trees are the same, but the nodes of the adder trees are more complex and slower than the nodes here.

4 Some Applications

We shall here start by looking at situations where the result of a rounding by truncation is to be utilized as an intermediate value subject to further processing.

The most obvious situation is where the result is to be added to something else, in which case the round-bit of a canonical encoding can be used as carry-in to a normal 2's complement adder, whether this is a redundant or a non-redundant adder. Note that adding two operands in canonical encoding, with result in the same encoding, can be performed by any kind of carry-completing 2's complement adder, taking one of the round-bits as carry-in to the addition, and leaving the other as the round-bit of the result.

For accumulation of many addends in canonical encoding, redundant adders as say carry-save adders directly apply. For each addition one round bit can be absorbed as carry-in, and the other kept as round bit of the result. At the end a final non-redundant result can be obtained by a carry-completing adder, absorbing the last round-bit.

In the case of an operand is to be subtracted, the inverted bit-pattern of the operand is fed to the adder together with the inverted round-bit as carry-in. Hence there is no need to convert from RN-Coding to regular 2's complement before any further additive processing.

The same applies if an RN-Coded value is to be used as the multiplier factor in a multiplication, and it is to be recoded to a higher radix, say 4 or 8, to reduce the height of the multiplier tree, or the number of cycles in an iterative multiplier. Here an RN-Coded operand directly recodes into such higher radices simply by grouping digits (which then also form RN-Codings).

4.1 Applications in Signal Processing

Although RN-Coding applies equally well to floating-point representations incorporating non-absorbed round-bits, as also suggested in [NMLE00], let us here think of use in high-speed fixed-point digital signal processing applications.

Two particular applications needing frequent rounding comes to mind here, calculation of inner products for filtering, and polynomial evaluations for approximation of standard functions. For the latter application, a very efficient way of evaluating a polynomial is to apply the *Horner Scheme*. Let $f(x) = \sum_{i=0}^{n} a_i x^i$ be such a polynomial approximation, then $f(x)$ is efficiently evaluated as

$$f(x) = (\cdots((a_n) * x + a_{n-1}) * x \cdots + a_1) * x + a_0,$$

where to avoid a growth in operand lengths, roundings are needed in each cycle of the algorithm. But here the round-bits can easily be absorbed in a subsequent arithmetic operation, only at the very end a regular conversion may be needed, but normally the result is to be used in some calculation, hence again a conversion may be avoided.

For inner product calculations, the most accurate result is obtained if accumulation is performed in double precision, it will even be exact when performed in fixed-point arithmetic. However, if double precision is not available it is essential that a fast and optimal rounding is employed during accumulation of the product terms.

4.2 An Extension to CORDIC algorithms

Although not based on RN-Codings, but along the same idea based on terms of alternating signs, we wish to decompose a number t as a sum

$$t = \sum_{i=0}^{\infty} b_i \arctan 2^{-i} \quad \text{for} \quad b_i \in \{-1, 0, 1\}$$

(i.e., as with the conventional CORDIC [Vol59] with double rotation – to handle the zeros), but with the additional constraint that:

– $b_n = -1 \Rightarrow$ the smallest $k > n$ such that $b_k \neq 0$ satisfies $b_k = +1$;
– $b_n = +1 \Rightarrow$ the smallest $k > n$ such that $b_k \neq 0$ satisfies $b_k = -1$.

Let us denote

$$t_n = \sum_{i=0}^{n-1} b_i \arctan 2^{-i}.$$

Thanks to the additional constraint, the error bound when we approximate t by t_n is $\arctan 2^{-n}$ instead of the usual, approximately twice larger bound $\sum_{k=n}^{\infty} \arctan 2^{-k}$. This is the equivalent, with the "base" $(\arctan 2^{-i})$ of the radix-2 (i.e., "base 2^{-i}") RN-coding. To perform rotations, we can store the angle decomposed in that "base", this will lead to more accurate rotations, provided we can easily compute the decomposition.

Lemma 7. *For any $x \geq 0$,*

$$\arctan x \leq 2 \arctan(x/2).$$

Proof. The result is immediate using the power series for $\arctan x$. □

Here is an algorithm for generating the decompositions. We assume that

$$|t| \leq \arctan 2^0 = \pi/4,$$

and assume we have already computed $b_0, b_1, \ldots, b_{n-1}$, and $t_n = \sum_{i=0}^{n-1} b_i \arctan 2^{-i}$.

First case: $t_n \leq t$

Assume $t - t_n \leq \arctan 2^{-n}$, we will show by induction that this will be satisfied. Let k be the largest integer $\geq n$ such that (induction hypothesis)

$$t_n + \arctan 2^{-k} \geq t.$$

We choose $b_k = 1$ and (if $n < k$) $b_n = b_{n+1} = \cdots = b_{k-1} = 0$. We easily get

1. $t_{k+1} = t_n + \arctan 2^{-k} \geq t$
   which implies that the next non-zero "digit" b_j, if any, will be -1;
2. $t_n + \arctan 2^{-k-1} \leq t$,
   which implies $-\arctan 2^{-k} + \arctan 2^{-k-1} \leq t - t_{k+1}$.

Hence, from the lemma $-\arctan 2^{-k-1} \leq t - t_{k+1} \leq 0$. which corresponds to the induction hypothesis.

Second case: $t_n \geq t$

This case is symmetrical to the previous one. We assume

$$t_n - \arctan 2^{-n} \leq t$$

(induction hypothesis). Let k be the largest integer $\geq n$ such that this condition is satisfied. Choose $b_k = -1$ and (if $n < k$) $b_n = b_{n+1} = \cdots = b_{k-1} = 0$, we then get as above:

1. $t_{k+1} \leq t$ (which implies that the next non-zero "digit" b_j, if any, will be $+1$);
2. $t_n - \arctan 2^{-k-1} \geq t$, which implies $0 \leq t - t_{k+1} \leq \arctan 2^{-k-1}$

which corresponds to the induction hypothesis.

5 Conclusions

Concentrating on binary RN-Coded operands, with the feature of rounding by truncation, we have shown how a simple encoding, based on the ordinary 2's complement representation, allows trivial conversion from 2's complement representation to RN-Coding, and a simple parallel prefix algorithm for conversion the other way. We have demonstrated how operands in this particular RN-Coding can be used at hardly any penalty in many standard calculations, while allowing rounding by a simple truncation. Thus in applications where many roundings are needed, it is possible to avoid the penalty of many intermediate log-time roundings, a single log-time rounding at the end is sufficient. As a small digression we have demonstrated how the idea of exploiting coefficients of alternating signs may be utilized in CORDIC algorithms, possibly an idea to be further explored.

References

[BK82] R. P. Brent and H. T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, C-31(3):260–264, March 1982.

[Boo51] A. D. Booth. A Signed Binary Multiplication Technique. *Q. J. Mech. Appl. Math.*, 4:236–240, 1951. Reprinted in [Swa80].

[KM05] P. Kornerup and J.-M. Muller. RN-Coding of Numbers: Definition and some Properties. In *Proc. IMACS'2005*, July 2005. Paris.

[NMLE00] A. Munk Nielsen, D. W. Matula, C. N. Lyu, and G. Even. An IEEE Compliant Floating-Point Adder that Conforms with the Pipelined Packet-Forwarding Paradigm. *IEEE Transactions on Computers*, 49(1):33–47, January 2000.

[Vol59] J. E. Volder. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959. Reprinted in [Swa80].

[Swa80] Earl E. Swartzlander, editor. *Computer Arithmetic, Vol I.* Dowden, Hutchinson and Ross, Inc., 1980. Reprinted by IEEE Computer Society Press, 1990.

Unifying Tests for Square Root

Michael Parks

Sun Microsystems
ieee754@yahoo.com

Abstract. We present an algorithm to generate good test cases for floating-point square root operations with regard to all four rounding modes from IEEE 754. The principal result is a new software implementation of existing square root test algorithms which not only tests all four modes at once, but significantly outpaces currently-available floating-point test software. The derivation of the test cases centers around an algorithm from number theory to solve congruences, and reveals an interesting connection between test data for the common round-to-nearest mode and the directed modes. A comparison of running times for computing test data will be made against currently-available software, along with remarks on efficiency.

1 Introduction

We revisit the problem of how to generate test cases for floating-point square roots using elementary number theory, taking into consideration all four rounding modes from IEEE Standard 754. The main result is a refreshed implementation in the C language for testing square roots which checks all four IEEE rounding modes essentially simultaneously. As well, our code significantly outperforms the square root testing program provided in the UCBTEST suite, due to a remarkable connection between test data for the usual round-to-nearest mode and the directed modes. After re-deriving the test in a unified manner, we give sample running times, software comparisons, and discuss the efficiency of the algorithm.

2 IEEE 754 Arithmetic

The IEEE Standard 754 [1] specifies that for a given precision n a set of representable numbers in normalized format be represented as $\{x : x = \pm 2^e(1+f)\}$, with an explicit sign bit, an integer exponent $e = \lfloor \log_2(|x|) \rfloor$ of limited range, and a fraction field $f = \sum_1^{n-1} b_j/2^j$ with bits $b_j = 0$ or 1. The nonzero fraction f is part of the significand $1+f$, which holds a number requiring n bits in its binary representation, and lies strictly between 0 and 1. In this note we will usually scale floating-point operands or results by powers of two in order to make them n-bit integers within the interval $[2^{n-1}, 2^n - 1]$, which will be referred to as the *fundamental range*. This note addresses operations upon normalized numbers exclusively, and does not require any mention of biased exponents, denormal numbers at the lowest edge of the exponent range, signed infinities and zeros, the NaN symbol for not-a-number, or exceptions, flags, and traps.

The standard mandates that every computed result of a basic mathematical operation $s = op(x)$ be rounded to n bits if $op(x)$ is not a machine-representable number, an infinity, or a NaN. The rounding mode controls the selection of the computed result, one of the two representable numbers nearest $op(x)$. In terms of the integer floor function $\lfloor t \rfloor$, ceiling function $\lceil t \rceil$, and nearest integer function $\lfloor t \rceil$ (breaking ties by selecting the even neighbor), the four IEEE rounding functions truncation, round-to-minus-infinity, round-to-nearest, and round-to-infinity are defined for $s > 0$ by the formulas

$$\mathrm{trunc}(s) = 2^{e(s)-n+1}\lfloor s/2^{e(s)-n+1}\rfloor$$

$$\mathrm{minf}(s) = 2^{e(s)-n+1}\lfloor s/2^{e(s)-n+1}\rfloor$$

$$\mathrm{near}(s) = 2^{e(s)-n+1}\lfloor s/2^{e(s)-n+1}\rceil$$

$$\mathrm{inf}(s) = 2^{e(s)-n+1}\lceil s/2^{e(s)-n+1}\rceil$$

(The precision n is implicit; for $s < 0$, swap the formulas for minf and inf). As our test cases are always positive, the trunc and minf rounding functions are mathematically identical henceforth, but in practice both should be explicitly tested since they could be implemented differently in logic.

The implementation of the floating-point square root instruction can be done in hardware but is usually done with a short software algorithm at the microcode level. The square root $\sqrt{x}$ of most every representable number x is not representable and so must be rounded. How square roots are computed using certain kinds of iterative algorithms will be reviewed briefly.

3 Algorithms for Square Root

In practice, $s = \sqrt{x}$ rounded to n bits in accordance with the rounding mode is computed using an iterative scheme, usually one based upon Newton's Method. An initial guess for $\sqrt{x}$, often looked up in a table, is improved upon using a formula until some desired accuracy limit is achieved, yielding a high-precision approximation to $\sqrt{x}$ when rounded, but in general, specialized analysis or an additional trick such as Tuckerman's condition is required to get the last bit or two right.

The classical iterative method is Heron's Rule, based upon the formula $f(s) = s^2 - x$ so that $f(\sqrt{x}) = 0$ and $f'(s) = 2s$. The iteration from Newton's Method $N(s) = s - f(s)/f'(s)$ is $s_{n+1} = s_n - f(s_n)/f'(s_n)$, or

$$s_{n+1} = \frac{s_n + x/s_n}{2}$$

which converges quadratically. Each step is relatively expensive in microcode, since each iteration costs 1 add, 1 shift, and 1 division which itself requires several multiplications and additions.

An alternate arrangement to calculate $\sqrt{x}$ is the reciprocal root algorithm in which $\sqrt{x}$ is well-approximated by the product of x and approximation to $1/\sqrt{x}$. An arrangement of Newton's method to produce an approximation to $1/\sqrt{x}$ can be derived by setting $f(s) = 1/s^2 - x$, so that $f(1/\sqrt{x}) = 0$ and $f'(s) = -2/s^3$. Newton's method is $N(s) = s - f(s)/f'(s) = s - (1/s^2 - x)/(-2/s^3) = s + (s/2)(1 - xs^2)$, a formula whose only division amounts to a simple register shift. Thus the iterative scheme

$$s_{n+1} = s_n + \frac{s_n(1 - xs_n^2)}{2}$$

converges to $1/\sqrt{x}$, again quadratically, and at a cheaper cost than the classical iteration: 3 multiplications, 2 adds, and 1 shift.

Iterative methods like Newton or Goldschmidt lead quickly to high-precision approximations to $\sqrt{x}$, but computing a correctly-rounded square root from them is not completely trivial. Roughly speaking, one should know $\sqrt{x}$ accurate to at least $2n$ bits in order to obtain a final approximation which then rounds correctly (see chapter 9 of Markstein [5]); this problem is reminiscent of the Table Maker's Dilemma [4]. If that approximation is separated from $\sqrt{x}$ by a *rounding boundary*, it will round incorrectly. This motivates a strategy to look for test cases: seek floating-point numbers x for which $\sqrt{x}$ lies near such a boundary. Although the introduction of fused multiply-addition allows for easier computation of correctly rounded results, floating-point algorithm errors have been identified in the past (see Parks [6]), and therefore testing remains an indispensable part of the design process. Before delving into the details of the test, we need a short algorithm for solving congruence equations which will be used in the remaining sections.

4 Hensel Lifting

This section contains most of the theoretical results needed in the following sections. We will rely strongly on the language of integer congruence, and write $A \equiv B \bmod C$ when $B - A$ is an integer multiple of C. The smallest A of this form lying between 0 and $C - 1$ is the remainder $A \bmod C$. As a special case, if $C = 2^j$, then the binary representations of integers A and B are identical in their last j bits.

Hensel Lifting is a technique from number theory which appears in more general contexts in a variety of forms, among them methods to solve equations in polynomial rings, and the field of p-adic analysis. Studies in analytic number theory such as Koblitz [3] may regard Hensel Lifting as the statement that Newton's Method converges in the p-adic metric, but to solve a few interrelated congruences which arise in the next section, a much simpler form with prime $p = 2$ will suffice.

Fact: Let $f(z) = z^2 - k$, and suppose odd z_j solves $f(z_j) \equiv 0 \bmod 2^j$. Put $z_{j+1} = z_j$ if $z_j^2 \equiv k \bmod 2^{j+1}$, and $z_{j+1} = 2^{j-1} - z_j$ otherwise. Then $f(z_{j+1}) \equiv 0 \bmod 2^{j+1}$.

Proof: Put $R = (z_j^2 - k)/2^j$, an integer by hypothesis. If R is even, then we have $z_{j+1}^2 - k = z_j^2 - k = 2^j R = 2^{j+1}(R/2)$, hence 2^{j+1} is a factor of $f(z_{j+1})$. If R is odd, then with $z_{j+1} = 2^{j-1} - z_j$ we have $z_{j+1}^2 - k = 2^{2j-2} - 2^j z_j + 2^j R = 2^{2j-2} + 2^{j+1}(R - z_j)/2$, and again 2^{j+1} is a factor of $f(z_{j+1})$.

Thus each solution z_j to $f(z_j) \equiv 0 \bmod 2^j$ admits a successor to $f(z_{j+1}) \equiv 0 \bmod 2^{j+1}$, and z_{j+1} is called a *lift* of z_j. Note that with regard to binary representations, we obtain z_{j+1} by either prepending a binary bit of 0 to z_j itself, or prepending a bit of 1 by taking the bitwise complement of z_j. The integer R is a kind of normalizing factor: provided k is a small integer, $2^j R$ lies near a perfect square.

Each number z_j which solves $z^2 \equiv k \bmod 2^j$ is called a 2-adic square root of k. We refrain from a full discussion of 2-adic arithmetic, but since, in the 2-adic system, numbers extend infinitely far to the left of the binary point, there are other solutions to the congruence. To record them, a property is worth noting.

Fact: If z satisfies the congruence $z^2 \equiv k \bmod 2^j$, then so do $2^{j-1}m \pm z$ for integers m.

Proof: $(2^{j-1}m \pm z)^2 = 2^{2j-2}m^2 \pm 2^j mz + z^2 = 2^j(2^{j-2}m^2 \pm mz) + z^2$, clearly also congruent to k with respect to 2^j.

Thus, 2-adic solutions other than z do exist; they can be obtained by prepending a 1 bit to z, or by taking bitwise complements. But not every integer k admits a solution to $z^2 \equiv k \bmod 2^j$ to begin with, a restriction follows:

Fact: If $j \geq 3$ and an odd z satisfies the congruence $z^2 \equiv k \bmod 2^j$, then $k \equiv 1 \bmod 8$.

Proof: Write $z = 2z' + 1$, then $z^2 = 4z'(z'+1) + 1$. Either z' or $z'+1$ is even, thus k has remainder 1 upon division by 8.

The next section will recast the problem of finding test data into the task of solving congruence equations of the form $z^2 - k \equiv 0 \bmod 2^j$ for four consecutive j. A recurrence derived from the Hensel Lifting construction in this section, plus some extra analysis to ensure that test arguments fit into n bits, allows solutions for all of them to be found in short order. Thus do we produce test data points for square root, and far more efficiently than previous efforts which addressed the to-nearest and directed modes of IEEE 754 separately. This is the advantage of the unified approach, both in the derivation in the next section, and in the software implementation discussed later.

5 Derivation of Test Data

This section shows how to generate rounding boundary cases for the square root function in a presentation similar to Parks [6] but in a unified manner so that all rounding directions are handled in tandem. In a software implementation, the algorithm below gives a speed advantage compared to the code in UCBTEST, as we will demonstrate conclusively.

The derivation is henceforth normalized so that for a fixed precision n, the algorithm will produce a test argument x of the form $2^{n-i}x'$ with x' in the fundamental range and $i = 0$ or 1. These choices ensure that x is machine-representable number, an integer of $2n - i$ bits whose binary form has $n - i$ trailing zeros. Since $\sqrt{x}$ is generally not a machine-representable number, the leading n bits in the binary expansion of $\sqrt{x}$ are followed by a binary point and a nonzero fraction between 0 and 1.

We seek test arguments whose square roots lie as close as possible to a rounding boundary. The characterization of such a boundary depends upon whether the rounding mode under consideration is to-nearest, handled shortly, or one of the directed modes.

For a directed mode, we wish to find an x (with $2^{2n-2} < x < 2^{2n}$ and $x = 2^{n-i}x'$), so that

$$\sqrt{x} = z \pm \epsilon$$

where z is an n-bit integer in the fundamental range, and tiny $\epsilon > 0$ barely deflects $\sqrt{x}$ from an n-bit integer, itself a rounding boundary of directed mode. Square to get

$$x = z^2 - k$$

where k is a small nonzero integer of either sign; $k = \pm 2\epsilon z - \epsilon^2$ is an integer because $z^2 - x$ is. By assumption, $x \equiv 0 \bmod 2^{n-i}$, hence we can find good test arguments by solving the congruences

$$z^2 \equiv k \bmod 2^{n-1} \quad \text{and} \quad z^2 \equiv k \bmod 2^n$$

for z, given precision n and choosing a small deflection parameter k. Solutions for even z can be found, but we will restrict z to be odd for the moment to be consistent with considerations for mode to-nearest.

Before delving into how to solve these congruence equations, we set up the problem to select test data for square root under mode to-nearest, since it leads directly to consideration of similar congruences.

For to-nearest, again normalizing so that $2^{2n-2} < x < 2^{2n}$ and $x = 2^{n-i}x'$, a rounding boundary case occurs just when $\sqrt{x}$ is near a midpoint of integers:

$$\sqrt{x} = y \pm (\frac{1}{2} - \epsilon)$$

where y is an n-bit integer in the fundamental range, and tiny $\epsilon > 0$ deflects $\sqrt{x}$ slightly from an integer midpoint $y + 1/2$ or $y - 1/2$. For such a case, we must have $y = \text{near}(\sqrt{x})$ and

$$4x = (2y + \sigma)^2 - k$$

for a small number $k = \pm 8\epsilon y + 4\epsilon - \epsilon^2$, where $\sigma = \text{sign}(k)$ is -1 or $+1$. Put $z = 2y + \sigma$, an odd integer in $[2^n + 1, 2^{n+1} - 1]$, so that $4x = z^2 - k$; then k must be an integer too. Since $4x \equiv 0 \bmod 2^{n+2-i}$ where $i = 0$ or 1, the search for good test arguments amounts to solving two congruences, with n and k given,

$$z^2 \equiv k \bmod 2^{n+1} \qquad \text{and} \qquad z^2 \equiv k \bmod 2^{n+2}$$

for an odd number z, and computing test argument x from z.

All of the congruences we wish to solve have the same form:

$$z^2 \equiv k \bmod 2^j$$

for exponents $j = n - 1, n, n + 1$, and $n + 2$. Hensel Lifting starting with a chosen $k \equiv 1 \bmod 8$ provides solutions for these exponents computed in turn. However, some careful analysis is necessary to produce test arguments x in the right range, and fortunately the correctly rounded results are provided automatically as a result of running the recurrence below.

```
HENSEL LIFTING ALGORITHM
select a small $k$ from $\{..., -7, 1, 9, ...\}$
take $z_3 = 1$ and $R_3 = (1 - k)/8$
for $j = 4, ..., n + 2$
    if $R_{j-1}$ is even, take
        $z_j = z_{j-1}$
        $R_j = R_{j-1}/2$
    otherwise take
        $z_j = 2^{j-2} - z_{j-1}$
        $R_j = 2^{j-4} + (R_{j-1} - z_{j-1})/2$
```

A couple of properties of the sequence $\{z_j\}$ are easily verified. First, z_j is odd for every j, and $1 \leq z_j \leq 2^{j-2} - 1$. Second, the recurrence ensures that $z_j^2 - k = 2^j R_j$, and so $z = z_j$ satisfies $z^2 \equiv k \bmod 2^j$ for each j, in particular for exponents $j = n - 1$ through $n + 2$.

With all four key congruences solved and z_{n-1}, z_n, z_{n+1}, and z_{n+2} at hand, the aforementioned properties do provide test arguments and the correctly rounded square roots for each mode. We discuss directed square root first, following Parks [6], using parentheses to denote expressions which do not suffer roundoff.

At most two test integers can be found using z_{n-1}. For $z = 2^{n-1} + z_{n-1}$, take

$$x = z^2 - k = 2^{n-1}(2^{n-1} + (R_{n-1} + 2z_{n-1}))$$

and for $z = 3 \times 2^{n-2} - z_{n-1}$, provided $z < \sqrt{2^{2n-1} - 2^{n-1}}$, take

$$x = 2^{n-1}(9 \times 2^{n-3} + (R_{n-1} - 3z_{n-1}))$$

For these x, $e(x) = 2n - 2$ precisely as wanted for test integers. The bound upon z_{n-1} allows us to estimate how often the second case is valid, about $4\sqrt{2} - 5 = 65.7\%$ of the time.

Also, at most two test integers can be found from z_n. For $z = 2^{n-1} + z_n$, provided $z > 2^{n-1/2}$, take

$$x = 2^n(2^{n-2} + (R_n + z_n))$$

and for $z = 2^n - z_n$, take

$$x = 2^n(2^n + (R_n - 2z_n))$$

For these x, $e(x) = 2n - 1$ as wanted for larger test integers. Knowing that $z_n \leq 2^{n-2}$ and hence $2^{n-1} \leq z \leq 3 \times 2^{n-2}$ provides an estimate of how often the fourth case produces a test integer; it works for about $3 - 2\sqrt{2} = 17.2\%$ of choices of k.

For the test pairs (x, z) created from z_{n-1} and z_n, the correctly rounded results are tested as follows.

$$\begin{aligned}&\text{when } k < 0, \text{ test for}\\ &\quad \operatorname{trunc}(\sqrt{x}) = z\\ &\quad \operatorname{minf}(\sqrt{x}) = z\\ &\quad \operatorname{inf}(\sqrt{x}) = z + 1\end{aligned}$$

$$\begin{aligned}&\text{when } k > 0, \text{ test for}\\ &\quad \operatorname{trunc}(\sqrt{x}) = z - 1\\ &\quad \operatorname{minf}(\sqrt{x}) = z - 1\\ &\quad \operatorname{inf}(\sqrt{x}) = z\end{aligned}$$

For to-nearest, we use z_{n+1} and z_{n+2} from the Hensel Lifting algorithm to locate test arguments and correctly rounded results; part of the material below appears in the UCBTEST source code, but our compact formula for test arguments x mimic those appearing in Kahan [2].

Consider $z = 2^n + z_{n+1}$, which solves $z^2 \equiv k \bmod 2^{n+1}$. Test argument x can be computed without roundoff or resorting to multiprecision arithmetic via

$$x = \frac{z^2 - k}{4} = 2^{n-1}(2^{n-1} + (R_{n+1} + z_{n+1}))$$

Note that $e(x) = 2n - 2$ as wanted; this x is in the right range. Provided $z < 2^{n+1/2}$ (equivalently $z_{n+1} < (\sqrt{2} - 1)2^n$), then $4x \equiv 0 \bmod 2^{n+1}$, so this x has the desired binary form. The equations $y = \operatorname{near}(\sqrt{x})$ and $z = 2y + \sigma = 2^n + z_{n+1}$ imply that the correctly rounded result to test for is

$$y = \operatorname{near}(\sqrt{x}) = 2^{n-1} + \frac{z_{n+1} - \sigma}{2}$$

From the bound $1 \le z_{n+1} \le 2^{n-1}$, infer that $2^n \le z \le 3 \times 2^{n-1}$, so on average this z should produce a test integer at a rate of $2(\sqrt{2}-1) = 82.8\%$.

Next consider $z = 2^{n+1} - z_{n+2}$, which solves $z^2 \equiv k \bmod 2^{n+2}$. Test argument x can be computed simply from the formula

$$x = \frac{z^2 - k}{4} = 2^n(2^n - (z_{n+2} - R_{n+2}))$$

again without higher precision or roundoff error; note that $e(x) = 2n-1$ as wanted. Provided $2^n\sqrt{2} < z$ or equivalently $z_{n+2} < (2-\sqrt{2})2^n$, then $4x \equiv 0 \bmod 2^{n+2}$, and hence x is a valid test integer. The equations $y = \text{near}(\sqrt{x})$ and $z = 2y + \sigma = 2^n + z_{n+1}$ imply that the correctly rounded result to check is

$$y = \text{near}(\sqrt{x}) = 2^n - \frac{z_{n+2} + \sigma}{2}$$

Since $2^n \le z \le 2^{n+1}$, this z yields a test integer about $2 - \sqrt{2} = 58.5\%$ of the time.

6 Implementation and Comparison

We have implemented all the tests in the previous section in a C language program `fastsqr.c`, and then compared the execution times to compute a large number of test data for the square root instruction in IEEE single and double precisions to the program `sqr.c` in the UCBTEST suite, which checks to-nearest mode only. If to-nearest mode is selected for scrutiny, the test data produced by our program are exactly the same as those produced by UCBTEST.

To set up a test run, the user may choose various parameters through conditional compilation, e.g. the precision via

```
#define PRECISION double
/* also float, extended */
```

the rounding modes under examination:

```
#define TEST_NEAREST
#define TEST_DIRECTED
```

and the largest deflection parameter, e.g.

```
#define MAXK 200000000
```

At processor speeds of 1 to 2 GHz, IEEE single precision with $n = 24$ can be tested quite thoroughly using our program for all IEEE modes in mere seconds, for $|k|$ up to about 2^{n-5}, beyond which the square roots lie relatively far from rounding boundaries anyway. For IEEE double precision with $n = 53$, we compare the execution time measured in seconds of our program, to compute test data for and check 10^8 square root boundary cases, against the results from `sqr.c` in Table 1.

mode(s)	`sqr.c`	`fastsqr.c`
nearest	**6443**	**560**
directed	NA	688
all modes	NA	750

Table 1. Software running times

The new program is an order of magnitude faster than `sqr.c`. The reason is rather specific to the way UCBTEST's square root checking program is implemented. It does not use the normalizing factor R_j from the recurrence at all, but instead relies essentially on computing each instance of the expression $z_j^2 - k$ via simulated multiprecision arithmetic, which evidently proves quite costly compared to our implementation. Note that the cost of switching rounding modes during the test execution for all four IEEE modes together is immaterial compared to the huge savings if testing to-nearest and the directed modes separately.

Moreover, the new scheme generates test cases with great efficiency. As we have shown, at least two test integers for directed modes are absolutely guaranteed, a third is produced $4\sqrt{2} - 5 = 65.7\%$ of the time, and another different test case is found $3 - 2\sqrt{2} = 17.2\%$ of the time. For to-nearest, a first test integer is produced $2(\sqrt{2} - 1) = 82.8\%$ of the time, and a second $2 - \sqrt{2} = 58.5\%$ of the time. By summing these results, for a given small k, in theory, on average approximately $3\sqrt{2} = 4.24$ integers should be produced in return for n steps of the Hensel recurrence. For a moderate test run of 10^8 smallest choices of k, our program computed and checked 424264156 cases, a ratio which agrees with $3\sqrt{2}$ to 6 decimal figures.

7 Conclusion

In a certain sense it should come as no surprise that the test data for mode to-nearest and the directed modes are related. Indeed, a test point for $\text{near}(\sqrt{x})$ with precision n is also a test point for $\text{trunc}(\sqrt{x}) = \text{minf}(\sqrt{x})$ and $\text{inf}(\sqrt{x})$ if precision $n + 1$ were considered. The mathematical core of the test program is therefore the same: an algorithm which builds test arguments, each a 2-adic square root of a small deflection parameter, one bit a time from right to left. As noted, a small amount of extra analysis is mandatory in order to ensure that the test arguments lie in an acceptable range. The software implementation for the algorithm amounts to a powerful single loop which in our implementation has turned out to be significantly faster than a currently-available test library. In addition, our code generates cases at quite an efficient rate, on average over four tests in exchange for one time through the recurrence. Passing these tests do not replace proofs of correctness, but they can be used to gain a high degree of confidence that a floating-point square root algorithm conforms to IEEE Standard 754. We hope to make our codes available in the future.

8 Appendix

Our derivation tacitly relied upon the fact that the square root of an n-bit floating-point number never lands exactly on a midpoint of adjacent floating-point numbers. Doubtless very well-known, but much more is true:

Fact: If x is a representable number of n bits and $\sqrt{x}$ does not fit exactly into n bits, then $\sqrt{x}$ does not fit exactly into any greater number of bits.

Proof: Assume by suitable scaling that $2^{2n-2} < x < 2^{2n}$, with $x = 2^{n-i}x'$ and $i = 0$ or 1, and suppose to the contrary that $\sqrt{x}$ fits into no fewer than $n + p$ bits with $p \geq 1$. Take integer $y = \text{trunc}(\sqrt{x})$, so that $\sqrt{x} = y + z/2^p$ for a nonzero odd integer z below 2^p. Square to obtain $2^{2p}x = (2^p y + z)^2$; the left term is even, the right term odd, a contradiction.

9 Acknowledgments

Douglas Priest and Neil Toda read several drafts of this note and contributed thoughtful observations and suggestions. The previous work of Professor W. Kahan and UCBTEST test code by K. C. Ng is gratefully noted. The concerns, notational suggestions, and comments by the sharp referees were appreciated.

References

1. IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute for Electrical and Electronics Engineers, New York, 1985.
2. William Kahan, A Test for Correctly Rounded SQRT, manuscript at the URL `http://www.cs.berkeley.edu/~wkahan/SQRTest.ps` .
3. Neal Koblitz, p-adic Numbers, p-adic Analysis, and Zeta-Functions, Graduate Texts in Mathematics, volume 58, Springer-Verlag, 1984.
4. Vincent Lefevre, Jean-Michel Muller, and Arnaud Tisserand. Towards correctly rounded transcendentals. IEEE Transactions on Computers, version 47 (11), pages 1235 to 1243, November 1998.
5. Peter Markstein, IA-64 and Elementary Functions: Speed and Precision, Hewlett-Packard Professional Books, Prentice Hall, 2000.
6. Michael Parks, Number-theoretic Test Generation for Directed Rounding, IEEE Transactions on Computers, Special Issue on Computer Arithmetic, version 49 (7), pages 651 to 658, July 2000.
7. Ping Tak Peter Tang, Testing Computer Arithmetic by Elementary Number Theory, Preprint MCS—P84—0889, Mathematics and Computer Science Division, Argonne National Laboratory, August 1989.
8. The U.C. Berkeley test suite, `http://www.netlib.org/fp/ucbtest.tgz` .

Part III

Abstracts of Posters

An Experimental Stochastic Lazy Floating-Point Arithmetic

Keith Briggs

BT Research
keith.briggs@bt.com

1 Introduction

We are concerned here with problems of the type "determine rigorously whether $x < y$ or not", where x and y are computed numbers. Such a calculation can in principle be solved by a computation which terminates in finite time, assuming x and y are not equal. Problems of this type occur in computational number theory and computational geometry.

The most important practical question in this field is: can we write a general-purpose software package which gives guaranteed results with an acceptably short computation time? Here are some approaches:

– Error analysis: perform (by hand) a complete analysis of the propagation of round-off error through the computation. Set the precision level of each step so that the final error is acceptably small. This method was used by some entries in the Many-digits friendly competition. But this is hardly feasible for a general-purpose software package.
– Exact real arithmetic: there have been many attempts at this - see the links at xrc. Essentially, these methods use lazy evaluation where the whole directed acyclic graph (DAG) representing the computation is stored, and intermediate results are re-evaluated as needed. Each step is performed with a bounded error, so the final error is bounded. The main problems in practice concern the storage required to store the DAG, and the fact that the error bounds are typically too pessimistic. In many cases, the minimum possible "graininess" of 1 bit (the smallest possible increase in precision per step) is larger than is actually needed to satisfy error bounds. In large calculations, this can result in unacceptable inefficiencies. Also, transcendental functions are difficult to make efficient in these methods.
– iRRAM: `http://www.informatik.uni-trier.de/iRRAM/`
– RealLib: `http://www.brics.dk/~barnie/RealLib/`

Experience shows that which approach performs best is highly problem-dependent.

2 The idea

The present method is an experimental stochastic variation of exact real arithmetic. The novelty is a way to avoid the 1-bit graininess mentioned above. It is assumed that arbitrary precision floating-point arithmetic with correct rounding is available (such as provided by mpfr). The ingredients are:

– The complete DAG is stored, as in my xrc.
– The intermediate value at each node is a floating-point approximation x and absolute error bound e. The interval $[x - e, x + e]$ always strictly bounds the correct result. This interval will be refined automatically as necessary.
– When evaluation is initiated, the increase in precision as we move away from the output node is probabilistic, controlled by a parameter α. The optimum value (i.e. that minimizing computation time) for α will depend on the problem, but the final results are independent of α. In essence, if $\alpha < 1$, then 1 bit is added with probability α. If $\alpha > 1$, then $\lfloor\alpha\rfloor$ bits are added. This allows tuning - for example, for the logistic map problem, we know that $\alpha = 1$ should be optimal, because the Liapunov exponent is such that one bit is lost on average per iteration.
– The results of evaluation-initiating functions are guaranteed correct.
– The string output function tries to print the requested number of digits correctly, but does not guarantee correctness.

The stochastic component is of course a heuristic. Why might this be reasonable? Why not compute errors bounds exactly, as iRRAM does? One answer is already given above - error bounds are always pessimistic. Another answer is that computing error bounds itself takes time. One might compare the way packet protocols are implemented - at the lowest level, packets are just transmitted, regardless of whether they might suffer collisions or otherwise get lost. So at this level the system is stochastic, and transmission is not reliable. It is made reliable by being wrapped in another layer (e.g. TCP), which detects when packets are lost and resends them if necessary. My mpfs software has a similar two-level design - the lower level is not reliable, but the upper layer (the only layer with which the user interacts), is reliable. (More accurately, the lower layer actually is reliable, but not necessarily precise enough. The upper layer ensures sufficient precision.)

Tests on a preliminary implemplentation indicate a performance comparable to hand-tuned codes for specific problems.

Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic

Marius Cornea and Cristina Anderson

Intel Corporation

The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [1] is being revised [2], and an important addition to the current text is the definition of decimal floating-point arithmetic [3]. This is aimed mainly to provide a robust, reliable framework for financial applications that are often subject to legal requirements concerning rounding and precision of the results in the areas of banking, telephone billing, tax calculation, currency conversion, insurance, or accounting in general. The lack of a good standard for decimal computations has led in the past to the existence of numerous proprietary software packages for this purpose, each with its own characteristics and capabilities. As part of the work described here, new algorithms were developed along with correctness proofs, and were used for the core of an implementation in software of the IEEE 754R decimal floating-point arithmetic. In the absence of hardware to perform IEEE 754R decimal floating-point operations, this new software package that will be fully compliant with the standard proposal should be an attractive option for various financial computations.

When using binary floating-point to approximate decimal calculations, rounding errors may occur when converting numerical values between their binary and decimal representations, or can accumulate differently in the course of a computation. For example, if the value 0.0007 is assigned to a C variable x of type float then 10000 * x will not have the value 7.0 (the IEEE 754 binary encoding of the result will be 0x40dfffff instead of 0x40e00000). Such errors are not acceptable in many cases of financial computations. The IEEE 754R standard proposal attempts to resolve such issues by defining all the rules for decimal floating-point arithmetic in a way that can be adopted and implemented on all computing systems in software, in hardware, or in a combination of the two. The software package whose core operations are described here is likely the first dedicated implementation of the IEEE 754R decimal floating-point arithmetic.

A decimal floating-point number n defined by the IEEE 754R standard proposal can be represented as $n = \pm C * 10^e$ where C is a positive integer coefficient with at most p decimal digits, and e is an integer exponent. The basic decimal floating-point formats denoted by decimal32, decimal64, and decimal128 are characterized respectively by precisions p = 7, 16, and 34, and exponent ranges of $[-95, +96]$, $[-383, +384]$, and $[-6143, +6144]$. Two encoding methods are considered in the IEEE 754R proposal [2]: the decimal encoding method and the binary encoding method. The new algorithms for the basic decimal floating-point arithmetic that are referred to here are equally applicable to both encodings, although they will be more efficient with the latter.

The most important decimal floating-point operations from a performance standpoint are conversions, addition, multiplication, division, and square root.

Conversions between decimal and binary formats are necessary at least for two reasons. First, if floating-point values are encoded in decimal format (string, BCD, IEEE 754R decimal encoding, or other) they need to be converted to binary before a software implementation of the decimal floating-point operation can take full advantage of the existing hardware for binary operations. This conversion is relatively easy to implement, and if possible it should exploit the available instruction-level parallelism. The opposite conversion has to be performed for example on results before writing them to memory or to disk, or for printing decimal numbers encoded in binary. The second reason for binary-to-decimal conversion could be for rounding decimal floating-point results to a pre-determined number of digits, if the exact result was calculated first in binary format. The straightforward method for this is to convert the exact result to decimal, round to the destination precision and then, if necessary, convert the coefficient of the final result back to binary. This step can be avoided completely if the coefficients are stored in binary or at least it can be made simpler if they are stored in a decimal format.

Property 1 serves this purpose and it is presented here because it was used extensively in the implementation of several basic decimal floating-point operations. It gives a precise way to 'cut off' x decimal digits from the lower part of an integer C when its binary representation is available, thus avoiding the need to convert C to decimal, remove the lower x decimal digits, and then convert the result back to binary. This property was applied to conversions from binary to decimal format as well as in the implementation of some of the most common decimal floating-point operations: addition (subtraction), multiplication, fused multiply-add, and in part, division.

For example if the decimal number $C = 123456789$ is available in binary and its six most significant decimal digits are required, Property 1 specifies precisely how to calculate the constant $k_3 \approx 10^{-3}$ so that

$\lfloor C{*}k_3 \rfloor = 123456$, with certainty, while using only the binary representation of C. (Note: the floor(x), ceiling(x), and fraction(x) functions are denoted here by $\lfloor x \rfloor$, $\lceil x \rceil$, and $\{x\}$ respectively.)

Property 1. *Let $C \in \mathbf{N}$ be a number in base $b = 2$ and $d_0{*}10^{q-1}+d_1{*}10^{q-2}+d_2{*}10^{q-3}+...+d_{q-2}{*}10^1+d_{q-1}$ its representation in base B=10, where $d_0, d_1, \ldots d_{q-1} \in \{0, 1, ..., 9\}$ and $d_0 \neq 0$.*

Let $x \in \{1, 2, 3, ..., q-1\}$ and $\rho = log_2 10$. If $y \in \mathbf{N}$, $y \geq \lceil \{\rho * x\} + \rho * q \rceil$ and k_x is the value of 10^{-x} rounded up to y bits (the subscript RP,y indicates rounding up y bits in the significand):

$k_x = (10^{-x})_{RP,y} = 10^{-x} * (1+\varepsilon)$, $0 < \varepsilon < 2^{-y+1}$

Then $\lfloor C * k_x \rfloor = d_0 * 10^{q-x-1} + d_1 * 10^{q-x-2} + d_2 * 10^{q-x-3} + ... + d_{q-x-2} * 10^1 + d_{q-x-1}$

The values k_x for all x of interest are pre-calculated and are stored as pairs (K_x, e_x), with K_x and e_x positive integers, and $k_x = K_x * 2^{-ex}$. This allows for implementations in the integer domain of several decimal floating-point operations, in particular addition, subtraction, multiplication, fused multiply-add, and certain conversions. For addition and multiplication an important property is that when rounding the exact result to p digits, the information necessary to determine whether the result is exact (in the IEEE 754 sense) or perhaps a midpoint is available in the product $C * k_x$ itself. This makes the rounding operation efficient, and relatively simple.

For the division operation, Property 1 is used only when an underflow is detected and the calculated quotient has to be shifted right a given number of decimal positions. For both division and square root, the algorithms for the general case are based on scaling the operands so as to bring the results into desired integer ranges, in conjunction with a few floating-point operations.

The algorithms and operations mentioned here represent the core of a new generic implementation in C of the decimal floating-point arithmetic specified in the IEEE 754R standard proposal. It was possible to compare the performance of the new software package for basic operations with that of the decNumber package contributed to GCC 4.2 [4]. The decNumber package represents the only other implementation of the IEEE 754R decimal floating-point arithmetic in existence at the present time. More than that, 'decNumber is a high-performance decimal arithmetic library in ANSI C, especially suitable for commercial and human-oriented applications' [5]. Extremely capable, it allows for integer, fixed-point, and floating-point decimal numbers and supports arbitrary precision values. However, its use in GCC 4.2 is limited to processing decimal numbers in the IEEE 754R formats. Table 1 shows the results of this comparison for basic 64-bit and 128-bit decimal floating-point operations, measured on the Intel®EM64t system running Microsoft®Windows Server 2003 Enterprise x64 Edition. The code was compiled with the Intel(R) C++ Compiler for Intel(R) EM64T-based applications, Version 9.0. The three values presented in each case represent minimum, median, and maximum values for a small data set covering operations from very simple (e.g. with operands equal to 0 or 1) to more complicated, e.g. on operands with 34 decimal digits in the 128-bit cases. For the new library further performance improvements can be attained by fine-tuning critical code sequences.

Operation	New Library [clk cycles]	decNumber Library [clk cycles]	DecNumber /New Lib.
64-bit ADD	14-140-241	99-1400-1741	4-10-14
64-bit MUL	21-120-215	190-930-1824	6-8-9
64-bit DIV	172-330-491	673-2100-3590	4-6-11
64-bit SQRT	15-288-289	82-16700-18730	7-58-107
128-bit ADD	16-170-379	97-2300-3333	4-13-14
128-bit MUL	19-300-758	95-3000-4206	5-10-18
128-bit DIV	153-250-1049	1056-2000-7340	4-8-9
128-bit SQRT	16-700-753	61-42000-51855	4-60-152

Table 1. New Decimal Floating-Point Library Performance vs. decNumber on EM64t (3.4 GHz Xeon). Minimum-median-maximum values are listed in sequence, after subtracting the call overhead.

It is also likely that properties and algorithms used in the new library for decimal floating-point arithmetic can be applied as well for a hardware implementation, with re-use of existing circuitry for binary operations. It is the authors' hope that the work described here will represent a step forward toward reliable as well as efficient implementations of the IEEE 754R decimal floating-point arithmetic.

References

[1] IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. IEEE, 1985
[2] IEEE 754R, Revised IEEE Standard 754-1985, `http://754r.ucbtest.org/drafts/754r.pdf` .
[3] Decimal Floating-Point: Algorism for Computers, M. Cowlishaw, 2003, 16th IEEE Symposium on Computer Arithmetic.
[4] Decimal Floating-Point Extension for C via decNumber, Jon Grimm, IBM, GCC Summit, 2005
[5] `http://www.alphaworks.ibm.com/tech/decnumber` .

Accurate Dot Products with FMA

Stef Graillat, Philippe Langlois, and Nicolas Louvet

Laboratoire LP2A, Université de Perpignan Via Domitia,
52, avenue Paul Alduy, F-66860 Perpignan Cedex, France
{graillat,langlois,nicolas.louvet}@univ-perp.fr

Abstract. The fused multiply and add (FMA) operation computes a floating point multiplication followed by an addition or a subtraction as a single floating point operation. Intel IA-64, IBM RS/6000 and PowerPC architectures implement this FMA operation. The aim of this poster is to study how the FMA improves the computation of dot product with classical and compensated algorithms. The latters double the accuracy of the former at the same working precision. Seven algorithms are considered. We present associated theoretical error bounds. Numerical experiments illustrate the actual efficiency in terms of accuracy and running time. We show that the FMA does not improve in a significant way the accuracy of the result whereas it increases significantly the actual speed of the algorithms.

The fused multiply and add (FMA) operation computes a floating point multiplication followed by an addition or a subtraction as a single floating point operation. This means that only one final rounding (to the working precision) error is generated by a FMA whereas two occur in the classical implementation of $x \times y + z$. Intel IA-64, IBM RS/6000 and PowerPC architectures implement this FMA operation. On the Itanium processor, the FMA operation enables a multiplication and an addition to be performed in the same number of cycles than one multiplication or one addition [4].

The FMA operation seems to be advantageous for both speed and accuracy. Indeed, it approximately halves the number of rounding errors in many numerical algorithms. This is the case for example within the computation of a dot product of two n-vectors where just n rounding errors occur instead of $2n - 1$ without FMA.

Moreover, it is well known that FMA yields an efficient computation of the rounding error generated by a floating point product. Such rounding error computation at the current working precision is a key task when implementing multi-precision libraries as double-double or quad-double ones [1] or even when designing compensated algorithms. Compensated algorithms implement inner computation of the rounding errors generated by the original algorithms and so provide more accurate results; [6, 5] are examples of compensated summation and dot product. The latter reference recently proved that these compensated implementations double the accuracy of the classical algorithm still running in the current working precision.

Here we study how the FMA can improve the computation of dot products in terms of accuracy and running time.

First, we consider the classical dot product computed at the working precision with or without FMA. We report the theoretical error analysis (worst case bounds) and some experimental results to show that the use of FMA only slightly improve the accuracy of the computed dot product, even if the number of rounding errors is halved.

Nevertheless, the accuracy provided by the classical dot product may not be sufficient when applied to ill conditioned dot products. Such cases appear for instance when computing residuals for ill conditioned linear systems. So we also consider accurate dot products whose computed result is as accurate as if computed in twice the working precision. Here we consider the classical dot product performed with double-double computation as it can be found in the XBLAS library [3] and the compensated dot product from [5] where the FMA is used to compute the rounding error generated by each product. We also present a new compensated dot product using a recent algorithm by Boldo and Muller [2] that computes the exact result of a FMA operation as the unevaluated sum of three floating point values. We present theoretical error bounds to prove that all these algorithms provide results as accurate as if computed in twice the working precision. Then we compare these implementations in terms of practical computing time to identify the best choice to double the computing precision. Our experimental results show that the compensated algorithms run both considerably faster than the one with double-double computation. Moreover, the most efficient approach to benefit from the availability of a FMA seems to be to compensate the rounding error generated by each (classical) product without using the FMA operation in the inner loop of the dot product.

Algorithm	Brief description
`Dot`	Dot product without `FMA`
`DotFMA`	Dot product with `FMA`
`Dot2FMA`	Compensated dot product with `FMA`
`DotThreeFMA`	Compensated `DotFMA` with `FMA`
`DotXBLASFMA`	XBLAS dot product with `FMA`

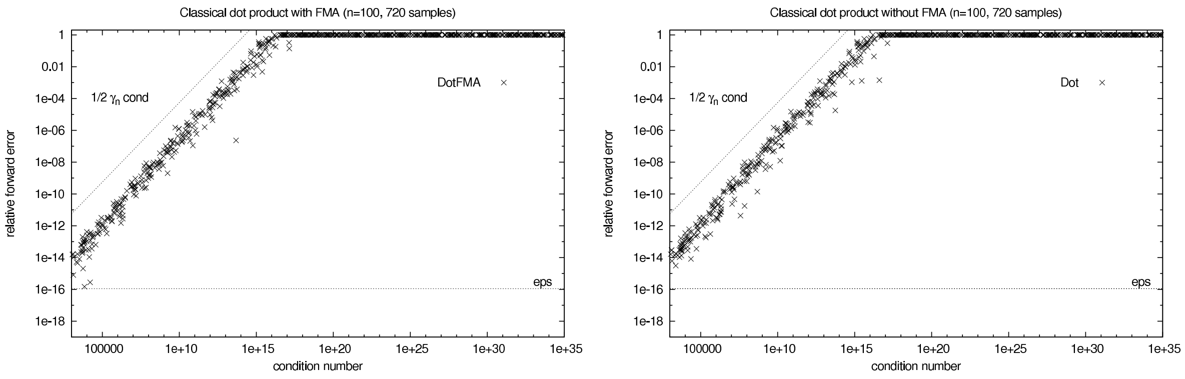


Fig. 1. Classical dot product with and without `FMA`: the `FMA` does not improve the actual accuracy.

	n	`DotFMA`	`Dot2FMA`	`DotThreeFMA`	`DotXBLASFMA`
Theoretical		1	10	19	22
Measured	50	1.0	1.4	2.3	8.24
	100	1.0	1.29	2.37	8.98
	1000	1.0	1.24	2.63	10.46
	10000	1.0	1.25	2.63	10.5
	100000	1.0	1.07	1.76	6.27

Table 1. Measured computing times (Itanium 2, 1600 MHz, Intel C++ Compiler v9.0). Using the `FMA` to compensate the multiplication rounding error (`Dot2FMA`) is the best choice to double the accuracy.

References

1. David H. Bailey. A Fortran-90 double-double library. Available at URL = `http://www.nersc.gov/~dhb/mpdist/mpdist.html`, 2001.
2. Sylvie Boldo and Jean-Michel Muller. Some functions computable with a fused mac. In IEEE, editor, *IEEE Symposium on Computer Arithmetic ARITH'17*, Cape Cod, Massachusetts, USA, June 2005.
3. Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.
4. Peter Markstein. *IA-64 and elementary functions. Speed and precision.* Hewlett-Packard Professionnal Books. Prentice-Hall PTR, 2000.
5. Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.
6. Michèle Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numer. Math.*, 19:400–406, 1972.

Online Generation of Extremal Rounding Test Sets for Floating Point Division

Jason S. Moore and David W. Matula[⋆]

Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275
{jmoore, matula}@engr.smu.edu

Testing rounding for division has proven to be no easy task as shown by the 1993 Pentium bug. IEEE standard single precision floating point division has 64 trillion possible input dividend, divisor pair values. That number of possible input values makes exhaustive testing unpractical. Therefore, carefully crafted ways of testing division must be found. The division extremal rounding test sets for round-to-nearest, RN_p, are sets of dividend, divisor input pairs whose infinitely precise quotients are extremely close to a midpoint between 2 p-bit numbers. Specifically, the quotient has a maximum number of like bits after the round bit [MM00, MM02, MM03]. This criteria for extremal rounding test cases for arithmetic operations and transcendental functions has been employed by several researchers using a variety of sophisticated number theoretic approaches [Ka87, LM99, Pa99, SZ05]. In order to be classified as a p-pit round-to-nearest extremal rounding input pair for division, the (n, d) pair can be characterized by three equivalent definitions below [MM00, MM02, MM03, MM06]:

- The distance of the quotient from the closest p-bit midpoint must satisfy $|\frac{n}{d} - \frac{i}{2^p}| < \frac{1}{2^{2p-1}}$
- The round bit of the quotient is followed by a run of *p-1* like bits that have the opposite value of the round bit.
- The proposed extremal rounding p-bit pair yields a quotient that is the closest rational approximation to the midpoint.

For $p = 5$, an example of an extremal round-to-nearest dividend, divisor pair of RN_p is 32 and 31 respectively. A similar definition characterizes the set of extremal rounding p-bit pairs for directed rounding RD_p [MM01].

We describe a C++ program that produces pseudo random p-bit extremal rounding test sets for $16 \le p \le 54$ for a 32-bit native machine and $16 \le p \le 64$ for a 64-bit native machine. The test sets can become very large as p increases, e.g. for $p = 28$ we obtain $|RN_{28}| = 93,035,551$ [MM06]. To obtain a pseudo random sample our program produces results similar to partitioning all of the pairs into blocks of size of order 1000 pseudo random extremal pairs each and randomly selecting a block. Importantly, the block is calculated efficiently on demand instead of being stored.

In order to calculate these values online, the following three values are needed: (i) the modular inverse of 3, (ii) a p-bit integer seed value (A), and (iii) the p-bit integer modular inverse of A. A is chosen randomly as follows. The $(p - 9)$ most significant bits are chosen randomly for A with the low order 9 bits $a_8a_7...a_0$ set to 1000 00001. Noting that $|(z2^8 + 1)(-z2^8 + 1)|_{2^{16}} = |(-z^2 2^{16} + 1)|_{2^{16}} = 1$, with $|.|_{2^{16}}$ denoting the standard residue modulo 2^{16}, by complementing leading bits we determine:

$$|A|_{2^{16}} = a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9 100000001,$$
$$|A^{-1}|_{2^{16}} = a'_{15}a'_{14}a'_{13}a'_{12}a'_{11}a'_{10}a'_9 100000001.$$

The nine least significant bits excluding third and first least significant bits are used to step through to get a block of 128 pseudo random multiplicative inverse pairs. At this point, $|A|_{2^{16}}$ and $|A^{-1}|_{2^{16}}$ are both 16-bits long. However for this to be useful, A^{-1} and 3^{-1} must initially be available at different bit lengths. Using the 16-bit,$|A^{-1}|_{2^{16}}$, the p-bit A^{-1} is calculated iteratively with quardradic convergence by using the formula, $|A^{-1}|_{2^{2n}} = (|A^{-1}|_{2^n} \times (2 - A \times |A^{-1}|_{2^n})) \bmod 2^{2n}$. Instead of storing, all possible $|3^{-1}|_{2^p}$, the single value $|3^{-1}|_{2^{64}}$ can be stored and all others of lesser precision can be found by truncation of leading bits. At this point 3 and $|3^{-1}|_{2^p}$ are used to step through successive examples. Note that from the multiplicative inverse pair (A, A^{-1}), we readily obtain a next inverse pair $(|3A|_{2^p}, ||3^{-1}|_{2^p}|A^{-1}|_{2^p}|_{2^p})$A is multiplied by 3 each time. This

⋆ These research results were obtained with support by SRC contract 1399

can be implemented by a shift and add. In order to step through the inverse, $|A^{-1}|_{2^p}$ is multiplied by $|3^{-1}|_{2^p}$ and evaluated mod 2^p. Therefore, each iteration requires only a shift, add, multiply, and AND operations.

Initially, two possible dividends, n and n', are calculated for use as a part of an extremal rounding pair. For each multiplicative inverse pair, (A, A^{-1}), up to a total of 12 extremal rounding pairs from RN_p and RD_p can be determined using only additions from either of th possible dividends via symmetric properties [MM01, MM06]. Whether none, one, or both of the dividends are used depends on the values of p, A, and A^{-1}. Let us define n and n' as $n = \frac{((2^p-A)\times A-1)A^{-1}}{2^p}$ and $n' = \frac{(2^p+A)(2^p-A^{-1})}{32}$ respectively. If p is odd, $2^{p-1}+1 \leq A^{-1} \leq 2^p - 1$ and n is either even or less than 2^p, $\frac{n}{A^{-1}}$ is a member of RN_p. $\frac{n'}{2^p-A^{-1}}$ is a member RN_p if p is odd, $1 \leq A^{-1} \leq 2^{p-1} - 1$, and n' is even or less than 2^p. If p is even, $2^{p-1}+1 \leq A \leq 2^p$, and n is even or less than 2^p, $\frac{n}{A}$ is a member of RN_p. $\frac{n'}{2^p-A}$ is a member RN_p if p is even, $1 \leq A \leq 2^{p-1} - 1$, and n' is even or less than 2^p. More details are given in [MM06].

[MM00, MM02, MM03, MM06] showed how to efficiently find extremal rounding dividend, divisor pairs and how effectively they could be used to check floating point division. We have presented here how to generate these values online with a time that is typically much less than the division operation that is being checked. More information and a program is available at [Ma06]. Currently, we are looking at how to extend this work to concurrently generate extremal rounding p-bit multiplier, multiplicand pairs for multiplication online.

References

[Ka87] W. Kahan, *Checking Whether Floating Point Division is Correctly Rounded*, monograph, April 11, 1987, `http://http.cs.berkeley.edu/~wkahan`

[LM99] V. Lefèvre, J.-M. Muller, A. Tisserand, "Towards Correctly Rounded Transcendentals", *Proc. 13th IEEE Symposium on Computer Arithmetic.* IEEE, 1997, pp. 132–137

[Ma06] D. W. Matula, `http://engr.smu.edu/~matula/extremal.html`

[Mc02] L. D. McFearin, "A p-Bit Model of Binary Floating Point Division and Square Root with Emphasis on Extremal Rounding Boundaries", Ph. D. Dissertation,Southern Methodist University, Dallas, Texas, May 2002.

[MM00] D. W. Matula, L. D. McFearin, "Number Theoretic Foundations of Binary Floating Point Division with Rounding", *Proceedings: Fourth Real Numbers and Computers*, April 2000, pp. 39–60.

[MM01] L. D. McFearin, D. W. Matula, "Generation and Analysis of Hard to Round Cases for Binary Floating Point Division", *Proc.* 15th *IEEE Symposium on Computer Arithmetic.* IEEE, 2001, pp. 119–127

[MM03] D. W. Matula, L. D. McFearin "A $p \times p$ Bit Fraction Model of Binary Floating Point Division and Extremal Rounding Cases". Journal of Theoretical Computer Science, 291:159–182, 2003.

[MM06] D. W. Matula, L. D. McFearin, "A Formal Method and Efficient Traversal Algorithm for Generating Testbenches for Verification of IEEE Standard Floating Point Division", *DATE 06.*,Mar. 2006, pp. 1134–1138

[Pa00] M. Parks, "Number-Theoretic Test Generation for Directed Rounding", *IEEE Transactions on Computers Volume 49.* IEEE, July 2000, pp. 651–658

[SZ05] D. Stehlé, P. Zimmermann, "Gal's Accurate Tables Method Revisited", *Proc.* 17th *IEEE Symposium on Computer Arithmetic.* IEEE, 2005, pp. 257–264

Author Index