Jean-Michel Muller

Elementary Functions

Algorithms and Implementation

Third Edition

Birkhäuser

Elementary Functions

Jean-Michel Muller

Elementary Functions

Algorithms and Implementation

Birkhäuser

Jean-Michel Muller
Laboratoire de l'Informatique du
Parallélisme (LIP)
École Normale Supérieure de Lyon CNRS
Lyon
France

ISBN 978-1-4899-7981-0 ISBN 978-1-4899-7983-4 (eBook)
DOI 10.1007/978-1-4899-7983-4

Library of Congress Control Number: 2016949097

Mathematics Subject Classification (2010): 65Y04, 65D15, 65D20, 68M07

Printed on acid-free paper

This book is published under the trade name Birkhauser, www.birkhauser-science.com
The registered company is Springer Science+Business Media LLC New York

Preface to the Third Edition

Since the publication of the second edition of this book, in 2006, several important changes have occurred in the domain of computer arithmetic.

First, a new version of the IEEE-754 Standard for Floating-Point Arithmetic was adopted in June 2008. This new version was merged with the previous binary (754) and "radix independent" (854) standards, resolved some ambiguities of the previous release, standardized the fused multiply–add (FMA) instruction, and included new formats (among them, the binary128 format, previously called "quad precision"). An appendix to this new IEEE 754-2008 standard also makes some important recommendations concerning the elementary functions.

New tools have been released that make much easier the work of a programmer eager to implement very accurate functions. A typical example is Sollya.[1] Sollya offers, among many interesting features, a certified supremum norm of the difference between a polynomial and a function. It also computes very good polynomial approximations with constraints (such as requiring the coefficients to be exactly representable in a given format). Another example is Gappa,[2] which simplifies the calculation of error bounds for small "straight-line" numerical programs (such as those designed for evaluating elementary functions), and makes it possible to use proof checkers such as Coq for certifying these bounds. FloPoCo[3] is a wonderful tool for implementing floating-point functions on FPGAs.

Research in this area is still very active. To cite a few examples: Harrison designed clever techniques for implementing decimal transcendental functions using the binary functions; Chevillard, Harrison, Joldes, and Lauter introduced a new algorithm for computing certified supremum norms of approximation errors; Johansson designed new algorithms for implementing functions in the "medium precision" range; Brunie et al. designed code generators for mathematical functions; several authors (especially de Dinechin) introduced hardware-oriented techniques targeted at FPGA implementation; Brisebarre and colleagues introduced methods for rigorous polynomial approximation.

[1] Available at http://sollya.gforge.inria.fr.

[2] Available at http://gappa.gforge.inria.fr.

[3] Available at http://flopoco.gforge.inria.fr.

Acknowledgments

I have benefited much from discussion with my colleagues and students. Especially, I am grateful to Jean-Claude Bajard, Jean-Luc Beuchat, Sylvie Boldo, Nicolas Brisebarre, Laurent-Stéphane Didier, Florent de Dinechin, Miloš Ercegovac, Stef Graillat, Guillaume Hanrot, Claude-Pierre Jeannerod, Mioara Joldes, Peter Kornerup, Christoph Lauter, Vincent Lefèvre, Nicolas Louvet, Erik Martin-Dorel, Guillaume Melquiond, Marc Mezzarobba, Adrien Panhaleux, Antoine Plet, Valentina Popescu, Bruno Salvy, Peter Tang, Arnaud Tisserand, and Serge Torres.

As usual, working with Birkhäuser's staff on the publication of a book is a pleasure.

This third edition was prepared in LaTeX on an Apple MacBook Pro laptop. I used the book document class of LaTeX with a few modifications, and the Xfig drawing tool, the TikZ package, GNU-Plot, or Maple for most figures. The text editor I used is TeXShop.

Lyon Jean-Michel Muller
April 2016

Preface to the Second Edition

Since the publication of the first edition of this book, many authors have introduced new techniques or improved existing ones. Examples are the *bipartite table method*, originally suggested by DasSarma and Matula in a seminal paper that led Schulte and Stine, and De Dinechin and Tisserand to design interesting improvements, the work on *formal proofs of floating-point algorithms* by (among others) John Harrison, David Russinoff, Laurent Théry, Marc Daumas and Sylvie Boldo, the design of *very accurate elementary function libraries* by people such as Peter Markstein, Shane Story, Peter Tang, David Defour and Florent de Dinechin, and the recently obtained results on the *table maker's dilemma* by Vincent Lefèvre. I therefore decided to present these new results in a new edition. Also, several colleagues and readers told me that a chapter devoted to multiple-precision arithmetic was missing in the previous edition. Chapter 7 now deals with that topic.

Computer arithmetic is changing rapidly. While I am writing these lines, the IEEE-754 Standard for Floating Point Arithmetic is being revised.[4] Various technological evolutions have a deep impact on determining which algorithms are interesting and which are not. The complexity of the architecture of recent processors must be taken into account if we wish to design high-quality function software: we cannot ignore the notions of pipelining, memory cache and branch prediction and still write efficient software. Also, the possible availability of a fused multiply-accumulate instruction is an important parameter to consider when choosing an elementary function algorithm.

A detailed presentation of the contents is given in the introduction. After a preliminary chapter that presents a few notions on computer arithmetic, the book is divided into three major parts. The first part consists of three chapters and is devoted to algorithms using polynomial or rational approximations of the elementary functions and, possibly, tables. The last chapter of the first part deals with multiple-precision arithmetic. The second part consists of three chapters, and deals with "shift-and-add" algorithms, i.e., hardware-oriented algorithms that use additions and shifts only. The last part consists of four chapters. The first two chapters discuss issues that are

[4]For information, see http://754r.ucbtest.org/.

important when accuracy is a major goal (namely, range reduction, monotonicity and correct rounding). The third one mainly deals with exceptions. The last chapter gives some examples of implementation.

Acknowledgments

I would like to thank all those who suggested corrections and improvements to the first edition, or whose comments helped me to prepare this one. Discussions with Nick Higham, John Harrison and William Kahan have been enlightening. Shane Story, Paul Zimmermann, Vincent Lefèvre, Brian Shoemaker, Timm Ahrendt, Nelson H.F. Beebe, Tom Lynch suggested many corrections/modifications of the first edition. Nick Higham, Paul Zimmermann, Vincent Lefèvre, Florent de Dinechin, Sylvie Boldo, Nicolas Brisebarre, Miloš Ercegovac, Nathalie revol read preliminary versions of this one. Working and conversing everyday with Jean-Luc Beuchat, Catherine Daramy, Marc Daumas, David Defour and Arnaud Tisserand has significantly deepened my knowledge of floating-point arithmetic and function calculation.

I owe a big "thank you" to Michel Cosnard, Miloš Ercegovac, Peter Kornerup and Tomas Lang. They helped me greatly when I was a young researcher, and with the passing years they have become good friends.

Working with Birkhäuser's staff on the publication of this edition and the previous one has been a pleasure. I have been impressed by the quality of the help they provide their authors.

Since the writing of the first edition, my life has changed a lot: two wonderful daughters, Émilie and Camille, are now enlightening my existence. This book is dedicated to them and to my wife Marie Laure.

This second edition was typeset in LaTeX on a DELL laptop. I used the book document style with a few modifications, and the Xfig drawing tool or Maple for most figures. The text editor I used is the excellent WinEdt software, by Aleksander Simonic (see http://www.winedt.com).

Lyon Jean-Michel Muller
April 2005

Preface to the First Edition

The elementary functions (sine, cosine, exponentials, logarithms…) are the most commonly used mathematical functions. Computing them quickly and accurately is a major goal in computer arithmetic. This book gives the theoretical background necessary to understand and/or build algorithms for computing these functions, presents algorithms (hardware-oriented as well as software-oriented), and discusses issues related to the accurate floating-point implementation of these functions. My purpose was not to give "cooking recipes" that allow to implement some given functions on some given floating-point systems, but to provide the reader with the knowledge that is necessary to build, or adapt algorithms to his or her computing environment.

When writing this book, I have had in mind two different audiences: *specialists*, who will have to design floating-point systems (hardware or software parts) or to do research on algorithms, and *inquiring minds*, who just want to know what kind of methods are used to compute the math functions in current computers or pocket calculators. Because of this, the book is intended to be helpful as well for postgraduate and advanced undergraduate students in computer science or applied mathematics as for professionals engaged in the design of algorithms, programs or circuits that implement floating-point arithmetic, or simply for engineers or scientists who want to improve their culture in that domain. Much of the book can be understood with only a basic grounding in computer science and mathematics: the basic notions on computer arithmetic that are necessary to understand are recalled in the first chapter.

The previous books on the same topic (mainly Hart et al. book *Computer Approximation* and Cody and Waite's book *Software Manual for the Elementary Functions*) contained many coefficients of polynomial or rational approximations of the elementary functions. I have included relatively few such coefficients here, firstly to reduce the length of the book – since I also wanted to present the shift-and-add algorithms –, and secondly because today it is very easy to obtain them using Maple or a similar system: my primary concern is to explain how they can be computed and how they can be used. Moreover, the previous books on elementary functions essentially focused on *software* implementations and polynomial or rational approximations, whereas now these functions are frequently implemented (at least partially) in hardware, using different methods (table-based methods or shift-and-add algorithms, such as CORDIC): I have wanted to show a large spectrum

of methods. Whereas some years ago a library providing elementary functions with one or two incorrect bits only was considered adequate, current systems must be much more accurate. The next step will be to provide *correctly rounded* functions (at least for some functions, in some domains), i.e., the returned result should always be the "machine number" that is closest to the exact result. This goal has already been reached by some implementations in single precision. I try to show that it can be reached in higher precisions.

Acknowledgments

Many people helped me during the process of writing this book. Many others gave me, during enlightening conversations, some views on the problem that deeply influenced me. It is not possible to cite everybody, but among those persons, I would especially like to thank:

- Jean-Marc Delosme, Warren Ferguson, Tomas Lang, Steve Sommars, Naofumi Takagi, Roger Woods and Dan Zuras, who volunteered to read parts of this book and gave me good advice, and Charles Dunham, who provided me with interesting information;
- my former and current students Jean-Claude Bajard, Catherine Billet, Marc Daumas, Yvan Herreros, Sylvanus Kla, Vincent Lefèvre Christophe Mazenc, Xavier Merrheim (who invented the tale presented at the beginning of Chapter 8), Arnaud Tisserand and Hong-Jin Yeh;
- the staff of the Computer Science Department and LIP laboratory at ENS Lyon.

Working with Birkhäuser on the publication of this book was a pleasure.

And of course, I thank my wife, Marie Laure, to whom this book is dedicated, for her patience and help during the preparation of the manuscript.

This book was typeset in LaTeX on a SUN workstation and an Apple Macintosh. I used the book document style. The text editors I used are Keheler's Alpha and GNU Emacs (Free Software Foundation).

Lyon Jean-Michel Muller
March 1997

Contents

List of Figures

List of Tables

Introduction 1

THIS BOOK IS DEVOTED to the computation of the elementary functions. Here, we call *elementary functions* the most commonly used mathematical functions: sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, sinh, cosh, tanh, $\sinh^{-1}$, $\cosh^{-1}$, $\tanh^{-1}$, exponentials, and logarithms (we should merely say "elementary transcendental functions": from a mathematical point of view, $1/x$ is an elementary function as well as e^x. We do not deal with the basic arithmetic functions in this book). Theoretically, the elementary functions are not much harder to compute than quotients: it was shown by Alt [5] that these functions are equivalent to division with respect to Boolean circuit depth. This means that, roughly speaking, a circuit can output n digits of a sine, cosine, or logarithm in a time proportional to $\log n$ (see also Okabe et al. [368], and Beame et al. [32]). For practical implementations, however, it is quite different, and much care is necessary if we want fast and accurate elementary functions.

This topic has already been dealt with, among others, by Cody and Waite [93], and Hart et al. [225], but at times those functions were implemented in *software* only and there was no standard for floating-point arithmetic. Since the Intel 8087 floating-point unit, elementary functions have sometimes been implemented, at least partially, in *hardware*, a fact that induces serious algorithmic changes. Even if now, for general-purpose computing, software is favored because of its versatility, there is still a clear need of hardwired functions for implementation on special-purpose architectures.

Furthermore, the emergence of high-quality arithmetic standards (such as the IEEE-754 standard for floating-point arithmetic), and the decisive work of mathematicians and computer scientists such as W. Kahan, W. Cody, H. Kuki, P. Markstein, and P. Tang have accustomed users to very accurate results. Twenty years ago a library providing elementary functions with one or two incorrect bits only was considered adequate [37], but current circuit or library designers must build algorithms and architectures that are guaranteed to be much more accurate (at least for general-purpose systems). A few libraries even offer *correctly rounded* functions: the returned result is always equal to the machine number nearest the exact result. Among the various properties that are desirable, one can cite the following:

- speed;
- accuracy;
- reasonable amount of resource (ROM/RAM, silicon area used by a dedicated hardware, even power consumption in some cases…);
- preservation of important mathematical properties such as *monotonicity*, and *symmetry*. As pointed out by Silverstein et al. [425], monotonicity failures can cause problems in evaluating divided differences;

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_1

- preservation of the direction of rounding: for instance, if the chosen rounding function is round toward $-\infty$ (see Section 2.1.2), the returned result must be less than or equal to the exact result. This is essential for implementing interval arithmetic;
- range limits: getting a sine larger than 1 may lead to unpleasant surprises, for instance, when computing [425]

$$\sqrt{1-\sin^2 x}.$$

Let us deal with the problem of accuracy. The 1985 version of the IEEE-754 standard for floating-point arithmetic [6]—which was the first version of that standard (see Section 2.1) greatly helped to improve the reliability and portability of numerical software. And yet it said nothing about the elementary functions. Concerning these functions, a standard cannot be widely accepted if some implementations are better than the standard. This means that when computing $f(x)$ we must try to provide the "best possible" result, that is, the *exact rounding* or *correct rounding* — see Chapter 2 for an explanation of "correct rounding" — of the exact result (when that result exists), for all possible input arguments.[1] This has already been mentioned in 1976 by Paul and Wilson [380]:

> *The numerical result of each elementary function will be equal to the nearest machine-representable value which best approximates (rounded or truncated as appropriate) the infinite precision function value for that exact finite precision argument for all possible machine-representable input operands in the legal domain of the function.*

As noticed by Agarwal et al. [2], correct rounding facilitates the preservation of monotonicity and, in round-to-nearest mode, symmetry requirements. And yet, for a few functions, correct rounding might prevent satisfying range limits (see Chapter 12 for an example). Also, correctly rounded functions are more difficult to implement. They may sometimes require very accurate intermediate computations. Consider for example the following number (represented in radix 2, with the exponent in radix 10).

$$1.11100100010110010110010100100110101111111100101001101 \times 2^{-10}$$

This number is exactly representable in the IEEE-754 double-precision format (see Chapter 2). Its radix-2 exponential is

$$\overbrace{1.00000000010100111111111100\cdots0011}^{53\text{ bits}}\,0\,\overbrace{111111111111\cdots111111111111}^{59\text{ ones}}010\cdots$$

which is very close to the middle of two consecutive floating-point numbers: deciding whether this value is above or below that middle (and hence, deciding what value should be returned in round-to-nearest mode) requires a very careful and accurate intermediate computation. A discussion on what could and/or should be put in a standardization of mathematical function implementation in floating-point arithmetic was given in [146]. The 2008 release of the IEEE-754 standard for floating-point arithmetic [245], in its appendix, now recommends (yet does not require) that a correctly rounded version of some functions should be available (see Chapter 12).

There is another difference between this book and those previously published which have dealt with the same topic: the latter have contained many coefficients of polynomial and/or rational approx-

[1] A usual objection to this is that most of the floating-point variables in a program are results of computations and/or measurements; thus they are not exact values. Therefore, when the least significant digit of such a floating-point number has a weight larger than π, its sine, cosine, or tangent have no meaning at all. Of course, this will be frequently true, but my feeling is that the designer of a circuit/library has no right to assume that the users are stupid. If someone wants to compute the sine of a very large number, he or she may have a good reason for doing this and the software/hardware must provide the best possible value.

imations of functions. Nowadays, software such as Maple[2] [78] or Sollya[3] [85] readily compute such coefficients with very good accuracy, requiring a few seconds of CPU time on a laptop. Therefore, the goal of this book is to present various algorithms and to provide the reader with the preliminary knowledge needed to design his or her own software and/or hardware systems for evaluating elementary functions.

When designing such a system, three different cases can occur, depending on the underlying arithmetic:

- the arithmetic operators are *designed specifically* for the elementary function system;
- the accuracy of the underlying arithmetic is *significantly higher* than the target accuracy of the elementary function system (for instance, binary32/single-precision functions are programmed using binary64/double-precision arithmetic, or binary64/double-precision functions are programmed using binary128 arithmetic);
- the underlying arithmetic is *not* significantly more accurate than the elementary function system (this occurs when designing routines for the highest available precision).

In the third case, the implementation requires much care if we wish to achieve last-bit accuracy. Some table-based algorithms have been designed to deal with this case, and a good knowledge of floating-point arithmetic allows one to avoid loosing accuracy in the intermediate calculations.

Chapter 2 of this book outlines several elements of computer arithmetic that are necessary to understand the following chapters. It is a brief introduction to floating-point arithmetic and redundant number systems. The reader accustomed to these topics can skip that chapter. That chapter cannot replace a textbook on computer arithmetic: someone who has to implement some elementary functions on a circuit or an FPGA may have to choose between different addition/multiplication/division algorithms and architectures. Several books devoted to computer arithmetic have been written by Swartzlander [439, 440, 442], Koren [277], Omondi [370], Parhami [378], Ercegovac and Lang [180], and Kornerup and Matula [282]. Division and square-root algorithms and architectures are dealt with in a book by Ercegovac and Lang [179]. The reader who wishes to deepen his or her knowledge of floating-point arithmetic can consult the recent Handbook of Floating-Point Arithmetic [356]. The reader can also find useful information in the proceedings of the IEEE Symposia on Computer Arithmetic, as well as in journals such as the *IEEE Transactions on Computers*, the *Journal of VLSI Signal Processing*, and the *Journal of Parallel and Distributed Computing*.

Aside from a few cases, the elementary functions cannot be computed exactly. They must be *approximated*. Most algorithms consist either of evaluating piecewise polynomial or rational approximations of the function being computed, or of building sequences that converge to the result.

Part 1 deals with the algorithms that are based on polynomial or rational approximation of the elementary functions, and/or tabulation of those functions. The classical theory of the approximation of functions by polynomials or rational functions goes back to the end of the nineteenth century. The only functions of one variable that can be computed using a finite number of additions, subtractions, and multiplications are polynomials. By adding division to the set of the allowed basic operations, we can compute nothing more than rational functions. As a consequence, it is natural to try to approximate the elementary functions by polynomial or rational functions. Such approximations were used much before the appearance of our modern electronic computers. However, it is only recently that we have been able to tackle more challenging problems such as finding best or nearly best approximations

[2]Maple is a registered trademark of Waterloo Maple Software.

[3]Sollya is a library for safe floating-point code development. It is especially targeted to the automatized implementation of mathematical floating-point libraries. It can be freely accessed at http://sollya.gforge.inria.fr.

with specific constraints on the coefficients such as being exactly representable in a given floating-point or fixed-point format. We will present these methods, and discuss the problem of *evaluating* the polynomials that have been chosen for approximating functions. Of special importance is the control of the numerical error due to the evaluation of the polynomials in finite precision arithmetic. Recent tools such as Gappa[4] are extremely useful for bounding that error and guaranteeing the bounds.

Accurate polynomial approximation to a function in a rather large interval may require a polynomial of large degree. For instance, approximating function $\ln(1+x)$ in $[-1/2, +1/2]$ with an error less than 10^{-8} requires a polynomial of degree 12. This increases the computational delay and may also induce problems of round-off error propagation, since many arithmetic operations need to be performed (unless a somewhat higher precision is used for the intermediate calculations). A solution to avoid these drawbacks is to use *tables*. However, this must be done with caution: possible cache misses due to large tables may totally destroy performance. Tabulating a function for all possible input values can be done for small word lengths (say, up to 20 bits). It cannot — at least with current technology — be done for larger word lengths: with 32-bit floating-point numbers, 16G-bytes of memory would be required for each function. For such word-lengths, one has to combine tabulation and polynomial (or rational) approximation. The basic method when computing $f(x)$ (after a possible preliminary range reduction), is to first locate in the table the value x_0 that is closest to x. Following this, $f(x)$ is taken as

$$f(x) = f(x_0) + \text{correction}(x, x_0),$$

where $f(x_0)$ is stored in the table, and $\text{correction}(x, x_0)$ — which is much smaller than $f(x_0)$ — is approximated by a low-degree polynomial. There are many possible compromises between the size of the table and the degree of the polynomial approximation. Choosing a good compromise may require to take into account the architecture of the target processor (in particular, the cache memory size [144]). This kind of method is efficient and widely used, yet some care is required to obtain very good accuracy.

When very high accuracy is required (thousands to billions of bits), the conventional methods are no longer efficient. One must use algorithms adapted to *multiple-precision* arithmetic. For instance, in 2002, Kanada's team from Tokyo University computed the first 1, 241, 100, 000, 000 decimal digits of π, using the following two formulas [16, 49]:

$$\pi = 48 \arctan \frac{1}{49} + 128 \arctan \frac{1}{57} - 20 \arctan \frac{1}{239} + 48 \arctan \frac{1}{110443}$$
$$\pi = 176 \arctan \frac{1}{57} + 28 \arctan \frac{1}{239} - 48 \arctan \frac{1}{682} + 96 \arctan \frac{1}{12943}.$$

This required 600 hours of calculation on a parallel Hitachi computer with 64 processors. At the time I am writing these lines, the record is around 13.3×10^{12} digits,[5] using formulas such as the following one, due to the Chudnovsky brothers [89]:

$$\frac{1}{\pi} = \frac{\sqrt{10005}}{4270934400} \sum_{k=0}^{\infty} (-1)^k \frac{(6k)!}{(k!)^3 (3k)!} \frac{13591409 + 545140134k}{640320^{3k}}.$$

[4] Gappa is a tool intended to help verifying and formally proving properties on numerical programs dealing with floating-point or fixed-point arithmetic. It can be freely obtained at http://gappa.gforge.inria.fr.

[5] See http://www.numberworld.org/y-cruncher/.

Part 2 is devoted to the presentation of *shift-and-add methods*, also called *convergence methods*. These methods are based on simple elementary steps, additions and shifts (i.e., multiplications by a power of the radix of the number system used), and date back to the seventeenth century. Henry Briggs (1561–1631), a contemporary of Napier (who discovered the logarithms), invented an algorithm that made it possible to build the first tables of logarithms. For instance, to compute the logarithm of x in radix-2 arithmetic, numerous methods (including that of Briggs, adapted to this radix) essentially consist of finding a sequence $d_k = -1, 0, 1$, such that

$$x \prod_{k=1}^{n} (1 + d_k 2^{-k}) \approx 1.$$

Then

$$\ln(x) \approx -\sum_{k=1}^{n} \ln(1 + d_k 2^{-k}).$$

The values $\ln(1 + d_k 2^{-k})$ are precomputed and stored. Another method belonging to the shift-and-add class is the CORDIC algorithm, introduced in 1959 by J. Volder and then generalized by J. Walther. CORDIC has great historical importance: as pointed out in [475], it has enabled pocket calculators to compute most elementary functions, making tables and slide rules obsolete. Nowadays, CORDIC is less frequently employed than table-based methods, but more recent developments on "redundant CORDIC" algorithms might one day change this situation. Moreover, CORDIC has a nice feature that is interesting for some applications: it directly computes functions of more than two variables such as *rotations* or *lengths* of 2-D vectors. Shift-and-add methods require less hardware than the methods presented in Part 1. Yet, they may be slower, and they are less versatile: they apply to some elementary functions only (i.e., functions f that satisfy some algebraic property allowing us to easily deduce $f(x + y)$ or $f(xy)$ from $f(x)$ and $f(y)$), whereas the methods based on polynomial approximation and/or tabulation can be used to design algorithms and architectures for any continuous function.

Another important step when computing an elementary function is *range reduction*. Most approximations of functions are valid in a small interval only. To compute $f(x)$ for any input value x, one must first find a number y such that $f(x)$ can easily be deduced from $f(y)$ (or more generally from an associated function $g(y)$), and such that y belongs to the interval where the approximation holds. This operation is called *range reduction*, and y is called the *reduced argument*. For many functions (especially the sine, cosine, and tangent functions), range reduction must be performed cautiously, it may be the most important source of errors.

The last part of this book deals with the problem of range reduction and the problem of getting correctly rounded final results, and give some examples of implementation.

To illustrate some of the various concepts presented in this introduction, let us look at an example. Assume that we use a radix-10 floating-point number system[6] with 4-digit significands,[7] and suppose that we want to compute the sine of $x = 88.34$.

The first step, range reduction, consists of finding a value x^* belonging to some interval I such that we have a polynomial approximation or a shift-and-add algorithm for evaluating the sine or cosine function in I, and such that we are able to deduce $\sin(x)$ from $\sin(x^*)$ or $\cos(x^*)$. In this example,

[6] Of course, in this book, we mainly focus on radix-2 implementations, but radix 10 leads to examples that are easier to understand. Radix 10 is also frequently used in pocket calculators.

[7] The word "significand" is more appropriate than the more frequently used word "mantissa". A system with p-digit significands will be said "of precision p."

assume that $I = [-\pi/4, +\pi/4]$. The number x^* is the only value $x - k\pi/2$ (k is an integer) that lies between $-\pi/4$ and $+\pi/4$. We can easily find $k = 56$, a consequence of which is $\sin(x) = \sin(x^*)$. After this, there are many various possibilities; let us consider some of them.

1. We simply evaluate $x^* = x - 56\pi/2$ in the arithmetic of our number system, $\pi/2$ being represented by its closest 4-digit approximation, namely, 1.571. Assuming that the arithmetic operations always return correctly rounded-to-the-nearest results, we get $X_1^* = 0.3640$. This gives one significant digit only, since the exact result is $x^* = 0.375405699485789\cdots$. Obviously, such an inaccurate method should be prohibited.
2. Using more digits for the intermediate calculations, we obtain the 4-digit number that is closest to x^*, namely, $X_2^* = 0.3754$.
3. To make the next step more accurate, we compute an 8-digit approximation of x^*; that is, $X_3^* = 0.37540570$.
4. To make the next step even more accurate, we compute a 10-digit approximation of x^*; that is, $X_4^* = 0.3754056995$.

During the second step, we evaluate $\sin(x^*)$ using a polynomial approximation, a table-based method, or a shift-and-add algorithm. We assume that, to be consistent, we perform this approximation with the same accuracy as that of the range reduction.

1. From X_1^* there is no hope of getting an accurate result: $\sin(X_1^*)$ equals $0.3560\cdots$, whereas the correct result is $\sin(x^*) = 0.3666500053966\cdots$.
2. From X_2^*, we get 0.3666. It is not the correctly rounded result.
3. From X_3^*, we get 0.366650006. If we assume an error bounded by 0.5×10^{-8} from the polynomial approximation and the round-off error due to the evaluation of the polynomial, and an error bounded by the same value from the range reduction, the global error committed when approximating $\sin(x^*)$ by the computed value may be as large as 10^{-8}. This does not suffice to round the result correctly: we only know that the exact result belongs to the interval [0.366649996, 0.366650016].
4. From X_4^*, we get 0.36665000541. If we assume an error bounded by 0.5×10^{-10} from the polynomial approximation and the possible round-off error due to the evaluation of the polynomial, and an error bounded by the same value from the range reduction, the global error committed when approximating $\sin(x^*)$ by the computed value is bounded by 10^{-10}. From this we deduce that the exact result is greater than 0.3666500053; we can now give the correctly rounded result, namely, 0.3667.

Although frequently overlooked, range reduction is the most critical point when trying to design very accurate libraries, especially in the case of trigonometric functions.

The techniques presented in this book will of course be of interest for the implementer of elementary function libraries or circuits. They will also help many programmers of numerical applications. If you need to evaluate a "compound" function such as $f(x) = \exp(\sqrt{x^2+1})$ in a given domain (say [0, 1]) only a few times and if very high accuracy is not a big issue, then it is certainly preferable to use the exp and $\sqrt{}$ functions available on the mathematical library of your system. And yet, if the same function is computed a zillion times in a loop and/or if it must be computed as accurately as possible, it might be better to directly compute a polynomial approximation to f using for instance the methods given in Chapter 3.

Introduction to Computer Arithmetic 2

2.1 Basic Notions of Floating-Point Arithmetic

The aim of this section is to provide the reader with some basic concepts of floating-point arithmetic, and to define notations that are used throughout the book. For further information, the reader is referred to the IEEE-754-2008 Standard on Floating-Point Arithmetic [245] and to our Handbook on Floating-Point Arithmetic [356]. Interesting and useful material can be found in Goldberg's paper [206] and Kahan's lecture notes [265]. Further information can be found in [55, 95, 102, 103, 180, 217, 232, 266, 277, 286, 370, 373, 378, 469, 477]. Here we mainly focus on the binary formats specified by the 2008 release of the IEEE-754 standard for floating-point arithmetic. The first release of IEEE 754 [6], that goes back to 1985, was a key factor in improving the quality of the computational environment available to programmers. Before the standard, floating-point arithmetic was a mere set of cooking recipes that sometimes worked well and sometimes did not work at all.[1]

2.1.1 Basic Notions

Everybody knows that a radix-β, precision-p floating-point number is a number of the form

$$\pm m \times \beta^e, \tag{2.1}$$

where m is represented with p digits in radix β, $m < \beta$, and e is an integer. However, being able to build trustable algorithms and proofs requires a more formal definition.

A floating-point format is partly[2] characterized by four integers:

- a *radix* (or *base*) $\beta \geq 2$;
- a *precision* $p \geq 2$ (p is the number of "significant digits" of the representation);
- two *extremal exponents* $e_{\min}$ and $e_{\max}$ such that $e_{\min} < e_{\max}$. In all practical cases, $e_{\min} < 0 < e_{\max}$.

[1] We should mention a few exceptions, such as some HP pocket calculators and the Intel 8087 coprocessor, that were precursors of the standard.

[2] Partly only, because bit strings must be reserved for representing exceptional values, such as the results of forbidden operations (e.g., $0/0$) and infinities.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_2

A finite floating-point number in such a format is a number x for which there exists at least one representation (M, e) that satisfies

$$x = M \cdot \beta^{e-p+1}, \tag{2.2}$$

where

- M is an integer of absolute value less than or equal to $\beta^p - 1$. It is called the *integral significand* of the representation of x;
- e is an integer such that $e_{\min} \leq e \leq e_{\max}$ is called the *exponent* of the representation of x.

We can now go back to (2.1), and notice that if we define $m = |M| \cdot \beta^{1-p}$ and $s = 0$ if $x \geq 0$, 1 otherwise, then

$$x = (-1)^s \cdot m \cdot \beta^e.$$

- m is called the *real significand* (or, more simply, the *significand* of the representation). It has one digit before the radix point, and at most $p - 1$ digits after; and
- s is the sign of x.

Notice that for some numbers x, there may exist several possible representations (M, e) or (s, m, e). Just consider the "toy format" $\beta = 10$ and $p = 4$. In that format $M = 4560$ and $e = -1$, and $M = 0456$ and $e = 0$ are valid representations of the number 0.456.

It is frequently desirable to require unique representations. In order to have a unique representation, one may want to *normalize* the finite nonzero floating-point numbers by choosing the representation for which the exponent is minimum (yet larger than or equal to $e_{\min}$). The obtained representation will be called a *normalized representation*. Two cases may occur.

- In general, such a representation satisfies $1 \leq |m| < \beta$, or, equivalently, $\beta^{p-1} \leq |M| < \beta^p$. In such a case, one says that x is a *normal* number.
- Otherwise, one necessarily has $e = e_{\min}$, and the corresponding value x is called a *subnormal* number (the term *denormal number* is often used too). In that case, $|m| < 1$ or, equivalently, $|M| \leq \beta^{p-1} - 1$. Notice that a subnormal number is of absolute value less than $\beta^{e_{\min}}$: subnormal numbers are very tiny numbers.

An interesting consequence of that normalization, when the radix β is equal to 2, is that the first bit of the significand of a normal number must always be "1", and the first bit of the significand of a subnormal number must always be "0". Hence if we have information[3] on the normality of x there is no need to store its first significand bit, and in many computer systems, it is actually not stored (this is called the "hidden bit" or "implicit bit" convention). Table 2.1 gives the basic parameters of the floating-point systems that have been implemented in various machines. Those figures have been taken from references [232, 265, 277, 370]. For instance, the largest representable finite number in the IEEE-754 double-precision/binary64 format [245] is

$$\left(2 - 2^{-52}\right) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308},$$

[3] In practice that information is encoded in the exponent field, see Section 2.1.6.

Table 2.1 *Basic parameters of various floating-point systems* (p, *the precision, is the size of the significand, expressed in number of digits in the radix of the computer system). The "+1" is due to the hidden bit convention. The binary32 and binary64 formats were called "single precision" and "double precision" in the* 1985 *release of the IEEE-754 standard.*

System	β	p	$e_{\min}$	$e_{\max}$	max. value
DEC VAX	2	24	−128	126	$1.7\cdots \times 10^{38}$
(D format)	2	56	−128	126	$1.7\cdots \times 10^{38}$
HP 28, 48G	10	12	−500	498	$9.9\cdots \times 10^{498}$
IBM 370	16	6 (24 bits)	−65	62	$7.2\cdots \times 10^{75}$
and 3090	16	14 (56 bits)	−65	62	$7.2\cdots \times 10^{75}$
IEEE-754 binary32	2	23+1	−126	127	$3.4\cdots \times 10^{38}$
IEEE-754 binary64	2	52+1	−1022	1023	$1.8\cdots \times 10^{308}$
IEEE-754 binary128	2	112+1	−16382	16383	$1.2\cdots \times 10^{4932}$
IEEE-754 decimal64	10	16	−383	384	$9.999\cdots 9 \times 10^{384}$

the smallest positive number is

$$2^{-1074} \approx 4.940656458412465 \times 10^{-324},$$

and the smallest positive normal number is

$$2^{-1022} \approx 2.225073858507201 \times 10^{-308}.$$

Arithmetic based on radix 10 has frequently been used in pocket calculators.[4] Also, it is used in financial calculations, and several decimal formats are specified by the 2008 version of IEEE 754. Decimal arithmetic remains an object of active study [114, 117, 183, 222, 453]. A Russian computer named SETUN [72] used radix 3 with digits $-1, 0$, and 1 (this is called the *balanced ternary system*). It was built[5] at Moscow University, during the 1960s [275]. Almost all other current computing systems use base 2. Various studies [55, 95, 286] have shown that radix 2 *with* the hidden bit convention gives better accuracy than all other radices (by the way, this does not imply that operations—e.g., divisions or square roots—cannot benefit from being done in a higher radix *inside* the arithmetic operators [181]).

2.1.2 Rounding Functions

Let us define a *machine number* to be a number that can be exactly represented in the floating-point system under consideration. In general, the sum, the product, and the quotient of two machine numbers is not a machine number and the result of such an arithmetic operation must be *rounded*.

In a floating-point system that follows the IEEE-754 standard, the user can choose a *rounding function* (also called *rounding mode*) from:

[4] A major difference between computers and pocket calculators is that usually computers do much computation between input and output of data, so that the time needed to perform a radix conversion is negligible compared to the whole processing time. If pocket calculators used radix 2, they would perform radix conversions before and after almost every arithmetic operation. Another reason for using radix 10 in pocket calculators is the fact that many simple decimal numbers such as 0.1 are not exactly representable in radix 2.

[5] See http://www.computer-museum.ru/english/setun.htm.

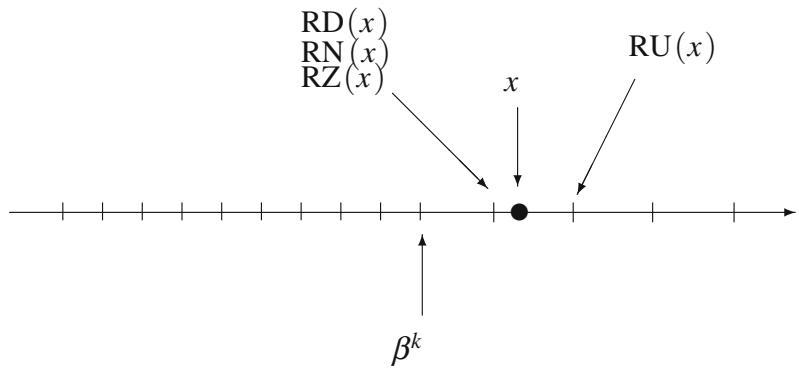


Figure 2.1 *Different possible roundings of a real number x in a radix-β floating-point system. In this example, $x > 0$.*

- rounding towards $-\infty$: $\mathrm{RD}(x)$ is the largest machine number less than or equal to x;
- rounding towards $+\infty$: $\mathrm{RU}(x)$ is the smallest machine number greater than or equal to x;
- rounding towards 0: $\mathrm{RZ}(x)$ is equal to $\mathrm{RD}(x)$ if $x \geq 0$, and to $\mathrm{RU}(x)$ if $x < 0$;
- rounding to nearest: $\mathrm{RN}(x)$ is the machine number that is the closest to x (if x is exactly halfway between two consecutive machine numbers, the default convention is to return the "even" one, i.e., the one whose last significand digit is even—a zero in radix 2).

This is illustrated using the example in Figure 2.1.

If the active rounding function is denoted by $\diamond$, and u and v are machine numbers, then the IEEE-754 standard [6, 109] requires that the obtained result should always be $\diamond(u \top v)$ when computing $u \top v$ ($\top$ is $+$, $-$, $\times$, or $\div$). Thus the system must behave as if the result were first computed *exactly*, with infinite precision, and then rounded. Operations that satisfy this property are called "correctly rounded" (or, sometimes, "exactly rounded"). There is a similar requirement for the square root. Such a requirement has a number of advantages:

- it leads to *full compatibility*[6] between computing systems: the same program will give the same values on different computers;
- many algorithms can be designed that use this property. Examples include performing large precision arithmetic [22, 231, 389, 423], designing "compensated" algorithms for evaluating with excellent accuracy the sum of several floating-point numbers [8, 264, 274, 386, 389, 404, 405], or making decisions in computational geometry [341, 387, 423];
- one can easily implement *interval arithmetic* [287, 288, 349], or more generally one can get lower or upper bounds on the exact result of a sequence of arithmetic operations;
- the mere fact that the arithmetic operations become fully specified makes it possible to elaborate formal proofs of programs and algorithms, which is very useful for certifying the behavior of numerical software used in critical applications [39, 41–44, 129, 133, 216–220, 315, 339, 340].

In radix-β, precision-p floating-point arithmetic, if an arithmetic operation is correctly rounded and there is no overflow or underflow[7] then the relative error of that operation is bounded by

[6] At least in theory: one must make sure that the order of execution of the operations is not changed by the compiler, that there are no phenomenons of "double roundings" due to the possible use of a wider format in intermediate calculations, and that an FMA instruction is called only if one has decided to use it.

[7] Let us say, as does the IEEE-754 standard, that an operation underflows when the result is subnormal *and* inexact.

$$\frac{1}{2}\beta^{1-p},$$

if the rounding function is round to nearest, and

$$\beta^{1-p}$$

with the other rounding functions.

Very useful algorithms that can be proved assuming correct rounding are the error-free transforms presented in Section 2.2.1 (the first ideas that underlie them go back to Møller [346]).

An important property of the various rounding functions defined by the IEEE-754 standard is that they are *monotonic*. For instance, if $x \leq y$, then $\mathrm{RN}(x) \leq \mathrm{RN}(y)$.

In the 1985 version of the IEEE-754 standard, there was no correct rounding requirement for the elementary functions, probably because it had been believed for many years that correct rounding of the elementary functions would be much too expensive. The situation has changed significantly in the recent years [125, 131, 136] and with the 2008 release of the IEEE-754 standard, correct rounding of some functions becomes recommended (yet not mandatory). These functions are:

$$\begin{gathered}
e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1, \\
\ln(x), \log_2(x), \log_{10}(x), \ln(1+x), \log_2(1+x), \log_{10}(1+x), \\
\sqrt{x^2+y^2}, 1/\sqrt{x}, (1+x)^n, x^n, x^{1/n} (n \text{ is an integer}), x^y, \\
\sin(\pi x), \cos(\pi x), \arctan(x)/\pi, \arctan(y/x)/\pi, \\
\sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x), \arctan(y/x), \\
\sinh(x), \cosh(x), \tanh(x), \sinh^{-1}(x), \cosh^{-1}(x), \tanh^{-1}(x).
\end{gathered}$$

We analyze the problem of correctly rounding the elementary functions in Chapter 12. Another frequently used notion is *faithful rounding*: a function is *faithfully rounded* if the returned result is always one of the two floating-point numbers that surround the exact result, and is equal to the exact result whenever this one is exactly representable. Faithful rounding cannot rigourously be called a *rounding* since it is not a deterministic function.

The availability of subnormal numbers (see Section 2.1.1) is a feature of the IEEE-754 standard that offers nice properties at the price of a slight complication of some arithmetic operators. It allows underflow to be gradual (see Figure 2.2). The minimum subnormal positive number in the IEEE-754 double-precision/binary64 floating-point format is

$$2^{-1074} \approx 4.94065645841246544 \times 10^{-324}.$$

In a floating-point system with correct rounding and subnormal numbers, the following theorem holds.

Theorem 1 (Sterbenz Lemma) *In a floating-point system with correct rounding and subnormal numbers, if x and y are floating-point numbers such that*

$$x/2 \leq y \leq 2x,$$

then $x - y$ is a floating-point number, which implies that it will be computed exactly, with any rounding function.

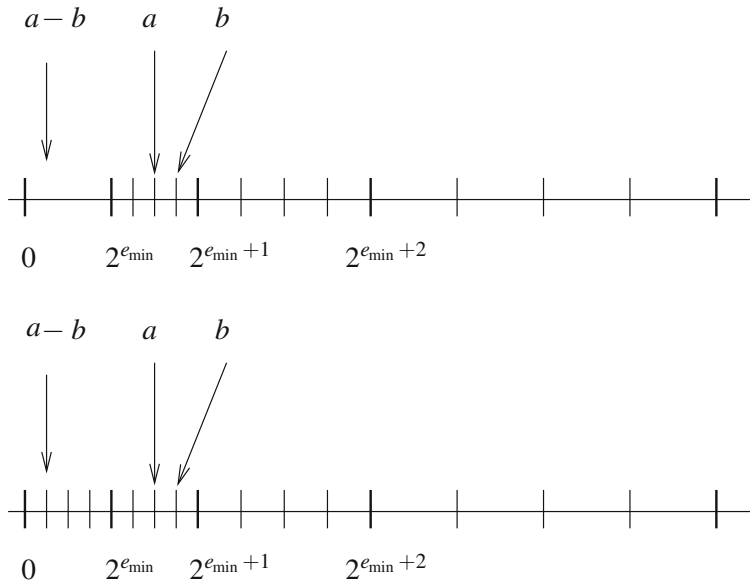


Figure 2.2 *Above is the set of the nonnegative, normal floating-point numbers (assuming radix* 2 *and* 2*-bit significands). In that set,* $a - b$ *is not exactly representable, and the floating-point computation of* $a - b$ *will return* 0 *with the round to nearest, round to* 0*, or round to* $-\infty$ *rounding functions. Below is the same set with subnormal numbers. Now,* $a - b$ *is exactly representable, and the properties* $a \neq b$ *and* $a \ominus b \neq 0$ *(where* $a \ominus b$ *denotes the computed value of* $a - b$*) become equivalent.*

This result is extremely useful when computing accurate error bounds for some elementary function algorithms.

The IEEE-754 standard also defines special representations for exceptions:

- NaN (Not a Number) is the result of an *invalid* arithmetic operation such as $0/0$, $\sqrt{-5}$, ∞/∞, $+\infty + (-\infty)$, …;
- $\pm\infty$ can be the result of an overflow, or the exact result of the division of a nonzero number by zero; and
- ± 0: there are two signed zeroes that can be the result of an underflow, or the exact result of a division by $\pm\infty$.

The reader is referred to [265, 356] for an in-depth discussion on these topics. Subnormal numbers and exceptions must not be neglected by the designer of an elementary function circuit and/or library. They may of course be used as input values, and the circuit/library must be able to produce them as output values when needed.

2.1.3 ULPs

If x is *exactly representable* in a floating-point format and is not an integer power of the radix β, the term ulp (x) (for *unit in the last place*) denotes the magnitude of the last significand digit of x. That is, if,

$$x = \pm x_0.x_1x_2\cdots x_{p-1} \times \beta^{e_x}$$

then $\text{ulp}(x) = \beta^{e_x - p + 1}$. Defining $\text{ulp}(x)$ for all reals x (and not only for the floating-point numbers) is desirable, since the error bounds for functions frequently need to be expressed in terms of ulps. There are several slightly different definitions in the literature [206, 217, 247, 267, 331, 373]. They differ when x is very near a power of β, and they sometimes have counterintuitive properties.

In this book, we will use the following definition.

Definition 1 (*ulp of a real number in radix-β, precision-p arithmetic of minimum exponent $e_{\min}$*) If x is a nonzero number, $|x| \in [\beta^e, \beta^{e+1})$, then $\text{ulp}(x) = \beta^{\max(e, e_{\min}) - p + 1}$. Furthermore, $\text{ulp}(0) = \beta^{e_{\min} - p + 1}$.

The major advantage of this definition (at least, in radix-2 arithmetic) is that in all cases (even the most tricky), rounding to nearest corresponds to an error of at most $1/2$ ulp of the real value. More precisely, we have

Property 1 *In radix 2, if X is a floating-point number, then*

$$|X - x| < \frac{1}{2}\, ulp\,(x) \Rightarrow X = \text{RN}(x).$$

(beware: that property does not always hold in radix-10 arithmetic)

Property 2 *For any radix,*

$$X = \text{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2}\, ulp\,(x).$$

We also have,

Property 3 *For any radix,*

$$X = \text{RN}(x) \Rightarrow |X - x| \leq \frac{1}{2}\, ulp\,(X).$$

(the difference with the previous property is that we have used the ulp of the computed value). Notice that $\text{ulp}(t)$ is a monotonic function of $|t|$: if $|t_1| \leq |t_2|$ then $\text{ulp}(t_1) \leq \text{ulp}(t_2)$. This has an interesting and useful consequence: if we know that the result of a correctly rounded (with round-to-nearest rounding function) arithmetic operation belongs to some interval $[a, b]$, then the rounding error due to that operation is bounded by

$$\frac{1}{2} \cdot \max\{\text{ulp}(a), \text{ulp}(b)\} = \frac{1}{2} \cdot \text{ulp}\,(\max\{|a|, |b|\}).$$

2.1.4 Infinitely Precise Significand

Implicitly assuming radix 2, we will extend the notion of significand to all real numbers as follows. Let x be a real number. If $x = 0$, then the infinitely precise significand of x equals 0, otherwise, it equals

$$\frac{x}{2^{\lfloor \log_2 |x| \rfloor}}.$$

The infinitely precise significand of a nonzero real number has an absolute value between 1 and 2. If x is a normal floating-point number, then its infinitely precise significand is equal to its significand.

2.1.5 Fused Multiply–Add Operations

The FMA instruction evaluates expressions of the form $ab + c$ with one rounding error instead of two (that is, if $\circ$ is the rounding function, $\mathrm{FMA}(a, b, c) = \circ(ab+c)$). That instruction was first implemented on the IBM RS/6000 processor [235, 348]. It was then implemented on several processors such as the IBM PowerPC [256], the HP/Intel Itanium [115], the Fujitsu SPARC64 VI, and the STI Cell, and is available on current processors such as the Intel Haswell and the AMD Bulldozer. More importantly, the FMA instruction is included in the 2008 release of the IEEE-754 standard for floating-point arithmetic [245], so that within a few years, it will probably be available on most general-purpose processors.

Such an instruction may be extremely helpful for the designer of arithmetic algorithms:

- it facilitates the exact computation of division remainders, which allows the design of efficient software for correctly rounded division [68, 113, 115, 265, 331, 333];
- it makes the evaluation of polynomials faster and—in general—more accurate: when using Horner's scheme,[8] the number of necessary operations (hence, the number of roundings) is halved. This is extremely important for elementary function evaluation, since polynomial approximations to these functions are frequently used (see Chapter 3). Markstein, and Cornea, Harrison, and Tang devoted very interesting books to the evaluation of elementary functions using the fused multiply–add operations that are available on the HP/Intel Itanium processor [115, 331];
- as noticed by Karp and Markstein [269], it makes it possible to easily get the exact product of two floating-point variables. More precisely, once we have computed the floating-point product π of two variables a and b (which is $\mathrm{RN}(ab)$ if we assume that the rounding function is round to nearest), one FMA operation suffices to compute the error of that floating-point multiplication, namely $ab - \pi$ (see Section 2.2.1).

And yet, as noticed by Kahan [265] a clumsy use (by an unexperienced programmer or a compiler) of a fused multiply–add operation may lead to problems. Depending on how it is implemented, function

$$f(x, y) = \sqrt{x^2 - y^2}$$

may sometimes return an NaN when $x = y$. Consider the following as an example:

$$x = y = 1 + 2^{-52}.$$

In binary64/double-precision arithmetic this number is exactly representable. The binary64 number that is closest to x^2 is

$$S = \frac{2251799813685249}{2251799813685248} = \frac{2^{51} + 1}{2^{51}},$$

and the binary64 number that is closest to $S - y^2$ is

$$-\frac{1}{2028240960365167042394725128 6016} = -2^{-104}.$$

[8] Horner's scheme consists in evaluating a degree-n polynomial $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ as $(\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})\cdots)x + a_0$. This requires n multiplications and n additions if we use conventional operations, or n fused multiply–add operations. See Chapter 5 for more information.

Hence, if the floating-point computation of $x^2 - y^2$ is implemented as $\mathrm{RN}(\mathrm{RN}(x^2) - y \times y)$, then the obtained result will be less than 0 and computing its square root will generate a NaN, whereas the exact result is 0. The problem does not occur if we do not use the FMA operation: the rounding functions are monotonic, so that if $|x| \geq |y|$ then the computed value of x^2 will be larger than or equal to the computed value of y^2.

2.1.6 The Formats Specified by the IEEE-754-2008 Standard for Floating-Point Arithmetic

Table 2.2 gives the widths of the various fields (significand, exponent) and the main parameters of the binary interchange formats specified by IEEE 754, and Table 2.3 gives the main parameters of the decimal formats. Let us describe the internal encoding of numbers represented in the *binary* formats of the Standard (for the internal encodings of decimal numbers, see [356]). The ordering of bits in the encodings is as follows. The most significant bit is the sign (0 for positive values, 1 for negative ones), followed by the exponent (represented as explained below), followed by the significand (with the hidden bit convention: what is actually stored is the "trailing significand," i.e., the significand without its leftmost bit). This ordering allows one to compare floating-point numbers as if they were sign-magnitude integers.

Table 2.2 *Widths of the various fields and main parameters of the binary interchange formats of size up to* 128 *bits specified by the 754-2008 standard.* [245]

IEEE 754-2008 name	binary16	binary32	binary64	binary128
Former name	N/A	Single precision	Double precision	Quad precision
Storage width	16	32	64	128
Trailing significand width	10	23	52	112
W_E, exponent field width	5	8	11	15
b, bias	15	127	1023	16383
Precision p	11	24	53	113
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$
$e_{\min}$	-14	-126	-1022	-16382
Largest finite number	65504	$2^{128} - 2^{104}$ $\approx 3.403 \times 10^{38}$	$2^{1024} - 2^{971}$ $\approx 1.798 \times 10^{308}$	$2^{16384} - 2^{16271}$ $\approx 1.190 \times 10^{4932}$
Smallest positive normal number	$2^{-14} \approx 6.104 \times 10^{-5}$	2^{-126} $\approx 1.175 \times 10^{-38}$	2^{-1022} $\approx 2.225 \times 10^{-308}$	2^{-16382} $\approx 3.362 \times 10^{-4932}$
Smallest positive number	$2^{-24} \approx 5.960 \times 10^{-8}$	2^{-149} $\approx 1.401 \times 10^{-45}$	2^{-1074} $\approx 4.941 \times 10^{-324}$	2^{-16494} $\approx 6.475 \times 10^{-4966}$

Table 2.3 *Main parameters of the decimal interchange formats of size up to* 128 *bits specified by the 754-2008 standard [245].*

IEEE-754-2008 name	decimal32	decimal64 (basic)	decimal128 (basic)
p	7	16	34
$e_{\max}$	$+96$	$+384$	$+6144$
$e_{\min}$	-95	-383	-6143

The exponents are represented using a *bias*. Assume the exponent is stored with W_E bits, and regard these bits as the binary representation of an unsigned integer N_e. Unless $N_e = 0$ (which corresponds to *subnormal* numbers and the two signed zeros), the (real) exponent of the floating-point representation is $N_e - b$, where $b = 2^{W_E-1} - 1$ is the *bias*. The value of that bias b is given in Table 2.2. N_e is called the *biased exponent*. All actual exponents from $e_{\min}$ to $e_{\max}$ are represented by N_e between 1 and $2^{W_E} - 2 = 1111\cdots110_2$. With W_E bits, one could represent integers from 0 to $2^{W_E} - 1 = 1111\cdots111_2$. The two extremal values 0 and $2^{W_E} - 1$, not needed for representing normal numbers, are used as follows.

- The extremal value 0 is reserved for subnormal numbers and ± 0. The bit encoding for a zero is the appropriate sign (0 for $+0$ and 1 for -0), followed by a string of zeros in the exponent field as well as in the significand field.
- The extremal value $2^{W_E} - 1$ is reserved for infinities and NaNs:
  - The bit encoding for infinities is the appropriate sign, followed by $N_e = 2^{W_E} - 1$ (i.e., a string of ones) in the exponent field, followed by a string of zeros in the significand field.
  - The bit encoding for NaNs is an arbitrary sign, followed by $2^{W_E} - 1$ (i.e., a string of ones) in the exponent field, followed by any bit string different from $000\cdots00$ in the significand field. Hence, there are several possible encodings for NaNs. This allows the implementer to distinguish between *quiet* and *signaling* NaNs (see [6, 245, 356] for a definition).

This encoding of binary floating-point numbers has a nice property: one obtains the successor of a floating-point number by considering its binary representation as the binary representation of an integer, and adding one to that integer.

2.1.7 Testing Your Computational Environment

The various parameters (radix, significand and exponent lengths, rounding functions…) of the floating-point arithmetic of a computing system may strongly influence the result of a numerical program. An amusing example of this is the following program, given by Malcolm [204, 330], that returns the radix of the floating-point system being used (beware: an aggressively "optimizing" compiler might decide to replace `((A+1.0)-A)-1.0` by `0`).

```
A := 1.0;
B := 1.0;
while ((A+1.0)-A)-1.0 = 0.0 do A := 2*A;
while  ((A+B)-A)-B <> 0.0 do B := B+1.0;
return(B)
```

Similar—yet much more sophisticated—algorithms are used in rather old inquiry programs such as MACHAR [98] and PARANOIA [270], that provide a means for examining your computational environment. Other programs for checking conformity of your computational system to the IEEE Standard for Floating Point Arithmetic are Hough's UCBTEST (available at http://www.netlib.org/fp/ucbtest.tgz), and a more recent tool presented by Verdonk, Cuyt and Verschaeren [462, 463].

2.2 Advanced Manipulation of FP Numbers

2.2.1 Error-Free Transforms: Computing the Error of a FP Addition or Multiplication

Let a and b be two precision-p floating-point numbers, and define $s = \mathrm{RN}(a+b)$, i.e., $a+b$ correctly rounded to the nearest precision-p floating-point number. It can be shown that if the addition of a and b does not overflow, then the error of that floating-point addition, namely $(a+b)-s$, is a precision-p floating-point number.[9] Interestingly enough, that error can be computed by very simple algorithms, as shown below.

Theorem 2 (Fast2Sum algorithm) *([148], and Theorem C of [275], p. 236). Assume the radix β of the floating-point system being considered is less than or equal to 3, and that the used arithmetic provides correct rounding with rounding to the nearest. Let a and b be floating-point numbers, and assume that the exponent of a is larger than or equal to that of b. Algorithm 1 below computes two floating-point numbers s and t that satisfy:*

- $s + t = a + b$ *exactly;*
- s *is the*[10] *floating-point number that is closest to* $a + b$.

Algorithm 1 The **Fast2Sum** algorithm [148].

$s \leftarrow \mathrm{RN}(a+b)$
$z \leftarrow \mathrm{RN}(s-a)$
$t \leftarrow \mathrm{RN}(b-z)$

Algorithm 1 requires that the exponent of a should be larger than or equal to that of b. That condition is satisfied when $|a| \geq |b|$. When we do not have preliminary information on a and b that allows us to make sure that the condition is satisfied, using Algorithm 1 requires a preliminary comparison of $|a|$ and $|b|$, followed by a possible swap of these variables. In most modern architectures, this comparison and this swap may significantly hinder performance, so that in general, it is preferable to use Algorithm 2 below, which gives a correct result whatever the ordering of $|a|$ and $|b|$ is.

Algorithm 2 The **2Sum** algorithm.

$s \leftarrow \mathrm{RN}(a+b)$
$a' \leftarrow \mathrm{RN}(s-b)$
$b' \leftarrow \mathrm{RN}(s-a')$
$\delta_a \leftarrow \mathrm{RN}(a-a')$
$\delta_b \leftarrow \mathrm{RN}(b-b')$
$t \leftarrow \mathrm{RN}(\delta_a+\delta_b)$

[9] Beware: that property is not always true with rounding functions different from RN. The error of a floating-point addition with one of these other rounding functions may not sometimes be exactly representable by a floating-point number of the same format.

[10] As a matter of fact there can be *two* such numbers (if $a+b$ is the exact middle of two consecutive floating-point numbers).

We have [41, 275, 346],

Theorem 3 *If a and b are normal floating-point numbers, then for any radix β, provided that no overflow occurs, the values returned by Algorithm 2 satisfy $a + b = s + t$.*

One can show that Algorithm 2 is optimal in terms of number of operations [280]. Algorithms 1 and 2 make it possible to compute the error of a floating-point addition. Interestingly enough, it is also possible to compute the error of a floating-point multiplication. Unless overflow occurs, Algorithm 3 below returns two values p and ρ such that p is the floating-point number that is closest to ab, and $p + \rho = ab$ exactly, provided that $e_a + e_b \geq e_{\min} + p - 1$ (where $e_{\min}$ is the minimum exponent of the floating-point format being used, e_a and e_b are the exponents of a and b, and p is the precision). It requires one multiplication and one fused multiply–add (FMA). Although I present it with the round-to-nearest function, it works as well with the other rounding functions.

Algorithm 3 The **Fast2MultFMA** algorithm.

$\pi \leftarrow \mathrm{RN}(ab)$;
$\rho \leftarrow \mathrm{RN}(ab - \pi)$

Performing a similar calculation without a fused multiply–add operation is possible [148] but requires 17 floating-point operations instead of 2. Some other interesting arithmetic functions are easily implementable when a fused multiply–add is available [45, 67, 253].

Algorithms 1, 2, and 3 can be used for building *compensated algorithms*, i.e., algorithms in which the errors of "critical" operations are computed to be later on "re-injected" in the calculation, in order to partly compensate for these errors. For example, several authors have suggested "compensated summation" algorithms (see for instance [261, 367, 386]). Another example is the following (notice that the first two lines are nothing but Algorithm 3):

Algorithm 4 Kahan's way to compute $x = ad - bc$ with fused multiply–adds.

$w \leftarrow \mathrm{RN}(bc)$
$e \leftarrow \mathrm{RN}(w - bc)$ // this operation is exact: $e = \hat{w} - bc$.
$f \leftarrow \mathrm{RN}(ad - w)$
$x \leftarrow \mathrm{RN}(f + e)$
return x

In [253], it is shown that in precision-p binary floating-point arithmetic, the relative error of Algorithm 4 is bounded by 2^{-p+1}, and that the error in ulps is bounded by $3/2$ ulps.

2.2.2 Manipulating Double-Word or Triple-Word Numbers

As we will see in Chapter 3, the elementary functions are very often approximated by polynomials. Hence, an important part of the function evaluation reduces to the evaluation of a polynomial. This requires a sequence of additions and multiplications (or a sequence of FMAs). However, when we want a very accurate result, it may not suffice to represent the coefficients of the approximating polynomial in the "target format."[11] Furthermore, to avoid a too large accumulation of rounding errors, it may

[11] Throughout the book, we call "target format" the floating-point format specified for the returned result, and "target precision" its precision.

sometimes be necessary to represent intermediate variables of the polynomial evaluation algorithm with a precision larger than the target precision. All this is easily handled when a wider floating-point format is available in hardware. When this is not the case, one can represent some high-precision variables as the unevaluated sum of two or three floating-point numbers. Such unevaluated sums are called "double-word" or "triple-word" numbers. Since the floating-point format used for implementing such numbers is almost always the binary64 format, previously called "double precision," these numbers are often called "double double" or "triple double" numbers in the literature.

Algorithms 1, 2, and 3 are the basic building blocks that allow one to manipulate double-word or triple-word numbers. For instance, Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) is shown in Algorithm 5 below.

Algorithm 5 Dekker's algorithm for adding two double-word numbers (x_h, x_ℓ) and (y_h, y_ℓ) [148].

if $|x_h| \geq |y_h|$ **then**
  $(r_h, r_\ell) \leftarrow \text{Fast2Sum}(x_h, y_h)$
  $s \leftarrow \text{RN}(\text{RN}(r_\ell + y_\ell) + x_\ell)$
else
  $(r_h, r_\ell) \leftarrow \text{Fast2Sum}(y_h, x_h)$
  $s \leftarrow \text{RN}(\text{RN}(r_\ell + x_\ell) + y_\ell)$
end if
$(t_h, t_\ell) \leftarrow \text{Fast2Sum}(r_h, s)$
return (t_h, t_ℓ)

The most accurate algorithm for double-word addition in Bailey's QD library, as presented in [317], is Algorithm 6 below.

Algorithm 6 The most accurate algorithm implemented in Bailey's QD library for adding two double-word numbers $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ [317].

$(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
$(t_h, t_\ell) \leftarrow \text{2Sum}(x_\ell, y_\ell)$
$c \leftarrow \text{RN}(s_\ell + t_h)$
$(v_h, v_\ell) \leftarrow \text{Fast2Sum}(s_h, c)$
$w \leftarrow \text{RN}(t_\ell + v_\ell)$
$(z_h, z_\ell) \leftarrow \text{Fast2Sum}(v_h, w)$
return (z_h, z_ℓ)

Assuming $|x_\ell| \leq 2^{-p} \cdot |x|$, where p is the precision of the binary floating-point arithmetic being used, one can show that the relative error of Algorithm 6 is bounded by

$$2^{-2p} \cdot \left(3 + 13 \cdot 2^{-p}\right),$$

(the bound is valid provided that $p \geq 3$, which always holds in practice).

Bailey's algorithm for multiplying a double-word number by a floating-point number is given below (here, we assume that an FMA is available, so that we can use Algorithm 3 to represent the product of two floating-point numbers by a double-word number).

Algorithm 7 The algorithm implemented in Bailey's QD library for multiplying a double-word number $x = (x_h, x_\ell)$ by a floating-point number y [317]. Here, we assume that an FMA instruction is available, so that we can use the Fast2MultFMA algorithm (Algorithm 3).

$(c_h, c_{\ell 1}) \leftarrow \text{Fast2MultFMA}(x_h, y)$
$c_{\ell 2} \leftarrow \text{RN}(x_\ell \cdot y)$
$(t_h, t_{\ell 1}) \leftarrow \text{Fast2Sum}(c_h, c_{\ell 2})$
$t_{\ell 2} \leftarrow \text{RN}(t_{\ell 1} + c_{\ell 1})$
$(z_h, z_\ell) \leftarrow \text{Fast2Sum}(t_h, t_{\ell 2})$
return (z_h, z_ℓ)

Assuming $|x_\ell| \leq 2^{-p} \cdot |x|$, one can show that the relative error of Algorithm 7 is bounded by

$$2 \cdot 2^{-2p}$$

Lauter [300] gives basic building blocks for a triple-word arithmetic. These blocks have been used for implementing critical parts in the CRLIBM library for correctly rounded elementary functions in binary64/double-precision arithmetic (see Section 14.4). For instance, here is one of Lauter's algorithms for adding two triple-word numbers and obtaining the result as a triple-word number.

Algorithm 8 An algorithm suggested by Lauter [300] for adding two triple-word numbers $a = (a_h, a_m, a_\ell)$ and $b = (b_h, b_m, b_\ell)$—a and b must satisfy the conditions of Theorem 4 below.

$(r_h, t_1) \leftarrow \text{Fast2Sum}(a_h, b_h)$
$(t_2, t_3) \leftarrow \text{2Sum}(a_m, b_m)$
$(t_7, t_4) \leftarrow \text{2Sum}(t_1, t_2)$
$t_6 \leftarrow \text{RN}(a_\ell + b_\ell)$
$t_5 \leftarrow \text{RN}(t_3 + t_4)$
$t_8 \leftarrow \text{RN}(t_5 + t_6)$
$(r_m, r_\ell) \leftarrow \text{2Sum}(t_7, t_8)$
return (r_h, r_m, r_ℓ)

Lauter showed the following result.

Theorem 4 *If Algorithm 8 is run in binary64 arithmetic with two input triple-word numbers* $a = (a_h, a_m, a_\ell)$ *and* $(b = (b_h, b_m, b_\ell)$ *satisfying:*

$$\begin{cases} |b_h| \leq (3/4) \cdot |a_h| \\ |a_m| \leq 2^{-\alpha_0} \cdot |a_h|, \text{ with } \alpha_0 \geq 4 \\ |a_\ell| \leq 2^{-\alpha_u} \cdot |a_m|, \text{ with } \alpha_u \geq 1 \\ |b_m| \leq 2^{-\beta_0} \cdot |b_h|, \text{ with } \beta_0 \geq 4 \\ |b_\ell| \leq 2^{-\beta_u} \cdot |b_m|, \text{ with } \beta_u \geq 1 \end{cases}$$

then the returned result (r_h, r_m, r_ℓ) *satisfies*

$$\begin{cases} r_h + r_m + r_\ell = ((a_h + a_m + a_\ell) + (b_h + b_m + b_\ell)) \cdot (1 + \epsilon), \\ \quad \text{with } |\epsilon| \leq 2^{-\min(\alpha_0+\alpha_u, \beta_0+\beta_u)-47} + 2^{-\min(\alpha_0, \beta_0)-98}, \\ |r_m| \leq 2^{-min(\alpha_0, \beta_0)+5} \cdot |r_h|, \\ |r_\ell| \leq 2^{-53} \cdot |r_m|. \end{cases}$$

The following is one of Lauter's algorithms for multiplying two double-word numbers and getting the result as a triple-word number.

Algorithm 9 An algorithm suggested by Lauter [300] for multiplying two double-word numbers $a = (a_h, a_\ell)$ and $b = (b_h, b_\ell)$, assuming a binary64 underlying arithmetic, and obtaining the product as a triple-word number—a and b must satisfy the conditions of Theorem 5 below. ADD22 is Algorithm 5.

$(r_h, t_1) \leftarrow \text{Fast2MultFMA}(a_h, b_h)$
$(t_2, t_3) \leftarrow \text{Fast2MultFMA}(a_h, b_\ell)$
$(t_4, t_5) \leftarrow \text{Fast2MultFMA}(a_\ell, b_h)$
$t_6 \leftarrow \text{RN}(a_\ell b_\ell)$
$(t_7, t_8) \leftarrow \text{ADD22}((t_2, t_3), (t_4, t_5))$
$(t_9, t_{10}) \leftarrow \text{2Sum}(\text{t}_1, \text{t}_6)$
$(r_m, r_\ell) \leftarrow \text{ADD22}((t_7, t_8), (t_9, t_{10}))$
return (r_h, r_m, r_ℓ)

Lauter showed the following result.

Theorem 5 *If Algorithm 9 is run in binary64 arithmetic with two input double-word numbers* $a = (a_h, a_\ell)$ *and* $b = (b_h, b_\ell)$ *satisfying* $|a_\ell| \leq 2^{-53}|a_h|$ *and* $|b_\ell| \leq 2^{-53}|b_h|$, *then the returned result* (r_h, r_m, r_ℓ) *satisfies*

$$\begin{cases} r_h + r_m + r_\ell = ((a_h + a_\ell) \cdot (b_h + b_\ell)) \cdot (1 + \epsilon), \\ \quad \textit{with } |\epsilon| \leq 2^{-149}, \\ |r_m| \leq 2^{-48} \cdot |r_h|, \\ |r_\ell| \leq 2^{-53} \cdot |r_m|. \end{cases}$$

2.2.3 An Example that Illustrates What We Have Learnt so Far

The following polynomial, generated by the Sollya tool (see Section 4.2) approximates function $\cos(x)$, for $x \in [-0.0123, +0.0123]$ (that domain is a tight enclosure of $[-\pi/256, +\pi/256]$), with an error less than 1.9×10^{-16}:

$$P(x) = 1 + a_2 x^2 + a_4 x^4,$$

where a_2 and a_4 are the following binary64/double-precision numbers:

$$\begin{cases} a_2 = -2251799813622611 \times 2^{-52} \approx -0.499999999986091 \\ a_4 = 1501189276987675 \times 2^{-55} \approx 0.04166637249080271 \end{cases}$$

Here we wish to have a tight upper bound on the error committed if we evaluate P as follows (Algorithm 10) in binary64/double-precision arithmetic. We wish to obtain the result as a double-word number (s_h, s_ℓ).

Algorithm 10 This algorithm returns an approximation to $P(x)$ as a double-word number (s_h, s_ℓ). We assume that an FMA instruction is available to compute $\text{RN}(a_2 + a_4 y)$ in line 2.

$y \leftarrow \text{RN}(x^2)$
$s_1 \leftarrow \text{RN}(a_2 + a_4 y)$
$s_2 \leftarrow \text{RN}(s_1 y)$
$(s_h, s_\ell) \leftarrow \text{Fast2Sum}(1, s_2)$
return (s_h, s_ℓ)

Since roundings are monotonic functions, we have

$$0 \le y \le \mathrm{RN}(0.0123^2).$$

Let us call b_y that bound, i.e.,

$$b_y = \frac{1395403955455759}{9223372036854775808}.$$

We have $\mathrm{ulp}\,(y) \le \mathrm{ulp}\,(b_y) = 2^{-65}$, therefore, since the computation of y is the result of a correctly rounded floating-point multiplication:

$$|y - x^2| \le \frac{1}{2}\,\mathrm{ulp}\,(b_y) = 2^{-66}. \tag{2.3}$$

So we have bounded the error committed at the first line of Algorithm 10. Let us now deal with the second line. We have

$$a_2 + a_4 y \in [a_2, a_2 + a_4 b_y]$$

therefore

$$\begin{aligned} s_1 &= \mathrm{RN}(a_2 + a_4 y) \\ &\in [\mathrm{RN}(a_2), \mathrm{RN}(a_2 + a_4 b_y)] = \left[a_2, -\frac{4503542848513793}{2^{53}}\right]. \end{aligned} \tag{2.4}$$

Thus $\mathrm{ulp}\,(s_1) \le \mathrm{ulp}\,(\max\{|a_2|, |a_2 + a_4 b_y|\}) = \mathrm{ulp}\,(|a_2|) = 2^{-54}$, from which we deduce

$$|s_1 - (a_2 + a_4 y)| \le 2^{-55}.$$

This gives

$$\begin{aligned} |s_1 - (a_2 + a_4 x^2)| &\le |s_1 - (a_2 + a_4 y)| + |(a_2 + a_4 y) - (a_2 + a_4 x^2)| \\ &\le 2^{-55} + a_4 \cdot |y - x^2|, \end{aligned}$$

from which we deduce, using (2.3),

$$|s_1 - (a_2 + a_4 x^2)| \le 2^{-55} + a_4 \cdot 2^{-66}. \tag{2.5}$$

We are now ready to tackle the third line of the algorithm. We have,

$$s_1 y \in \left[a_2 y, -\frac{4503542848513793}{2^{53}} \cdot y\right],$$

which implies,

$$s_1 y \in \left[a_2 \cdot b_y, 0\right],$$

therefore,

$$s_2 = \mathrm{RN}(s_1 \cdot y) \in \left[\mathrm{RN}\left(a_2 \cdot b_y\right), 0\right],$$

therefore,

$$s_2 \in \left[-\frac{5581615821667775}{2^{66}}, 0 \right]. \tag{2.6}$$

from this we deduce that $\text{ulp}(s_2) \leq 2^{-66}$, which implies

$$|s_2 - s_1 y| \leq 2^{-67}. \tag{2.7}$$

We now have

$$\begin{aligned} |s_2 - (a_2x^2 + a_4x^4)| &\leq |s_2 - s_1 y| + |s_1 y - s_1 x^2| + |s_1 x^2 - (a_2x^2 + a_4x^4)| \\ &\leq 2^{-67} + |s_1| \cdot |y - x^2| + x^2 \cdot |s_1 - (a_2 + a_4 y)| \\ &\leq 2^{-67} + |a_2| \cdot 2^{-66} + 0.0123^2 \cdot (2^{-55} + a_4 \cdot 2^{-66}) \end{aligned}$$

using (2.3), (2.4), and (2.5).

Finally, (2.6) implies $|s_2| < 1$, therefore Algorithm Fast2Sum is legitimately used at line 4 of the algorithm, so that $s_h + s_\ell = 1 + s_2$. We can therefore deduce a bound on the error committed when evaluating polynomial P using Algorithm 10:

$$\begin{aligned} |(s_h + s_\ell) - P(x)| &\leq 2^{-67} + |a_2| \cdot 2^{-66} + 0.0123^2 \cdot (2^{-55} + a_4 \cdot 2^{-66}) \\ &\leq 1.77518 \times 10^{-20}. \end{aligned} \tag{2.8}$$

The obtained bound (2.8) is rather tight: for instance, if we apply Algorithm 10 to the input value $x = 1772616811707781/2^{57}$, the evaluation error is 1.76697×10^{-20}.

The "toy" example we have considered here can be generalized to the evaluation of polynomials of larger degree, possibly using different evaluation schemes (see Chapter 5): the underlying idea is to compute interval enclosures of all intermediate variables, which allows to compute bounds on the rounding errors of the arithmetic operations. It can be adapted to compute relative error bounds instead of absolute ones. However, the idea of computing evaluation errors as we just have done here has some limitations:

- the process was already tedious and error prone with our toy example. In practical cases (degrees of polynomials that can be as large as a few tens, with some coefficients that can be double-word numbers), it may become almost impractical. Furthermore, to avoid inherent overestimations of enclosures that occur in interval arithmetic, one may need to split the input domain into many subintervals and redo the calculation for each of them;
- as we will see in Chapter 5, when some parallelism is available on the target processor (pipelined operators, several FPUs)—which is always the case with recent processors—many different evaluation schemes are possible (when the degree of the polynomial is large, the number of possible schemes is huge). In general, choosing which evaluation scheme will be implemented results from a compromise between the latency or throughput and accuracy. To find a good compromise, one may wish to get, in reasonable time, a tight bound on the evaluation error for several tens of evaluation schemes;
- if the function we are implementing is to be used in critical applications, one needs confidence in the obtained error bounds, which is not so obvious when they are derived from long and tedious calculations. One may even wish *certified* error bounds.

These remarks call for an automation of the calculation of error bounds for small "straight-line" numerical programs (such as those used for evaluating elementary functions), and for the possibility of using proof checkers for certifying these bounds. These needs are fulfilled by the Gappa tool, presented in the next section.

2.2.4 The GAPPA Tool

Thanks to the IEEE-754 standard, we now have an accurate definition of floating-point formats and operations. This allows the use of formal proofs to verify pieces of mathematical software. For instance, Harrison used HOL Light to formalize floating-point arithmetic [217] and check floating-point trigonometric functions [218] for the Intel-HP IA64 architecture. Russinoff [406] used the ACL2 prover to check the AMD-K7 Floating-Point Multiplication, Division, and Square Root instructions. Boldo, Daumas and Théry use the Coq proof assistant to formalize floating-point arithmetic and prove properties of arithmetic algorithms [42, 315].

The Gappa tool [128, 339] can be downloaded at http://gappa.gforge.inria.fr. It was designed by Melquiond to help to prove properties of small (up to a few hundreds of operations) yet complicated floating-point programs. Typical useful properties Gappa helps to prove are the fact that some value stays within a given range (which is important in many cases, for instance if we wish to guarantee that there will be no overflow), or that it is computed with a well-bounded relative error. The paper [134] explains how Gappa has been used to certify functions of the CRLIBM library of correctly rounded elementary functions. Gappa uses interval arithmetic, a database of rewriting rules, and hints given by the user to prove a property, and generates a formal proof that can be mechanically checked by an external proof checker. This was considered important by the authors of CRLIBM: as explained by de Dinechin et al. [134], in the first versions of the library, the complete paper and pencil proof of a single function required tens of pages, which inevitably cast some doubts on the trustability of the proof.

The following Gappa file automatically computes a bound on the error committed when evaluating the polynomial P of the previous section using Algorithm 10, with an input value in $[-0.0123, +0.0123]$. It is made up of three parts: the first one describes the numerical algorithm, the second one describes the exact value we are approximating, and the third one describes the theorem we wish to prove. For more complex algorithms, we may need a fourth part that describes hints given to Gappa.

```
@RN = float<ieee_64,ne>;
# defines RN as round-to-nearest in binary64 arithmetic

x = RN(xx);
a2 = -2251799813622611b-52;
a4 = 1501189276987675b-55;

# description of the program

y RN = x * x;
# now, we describe the action of the FMA
s1beforernd = a2 + a4*y;
s1 = RN(s1beforernd);
s2 RN = s1*y;
s = 1 + s2; # no rounding: Fast2Sum is an exact transformation

# description of the exact value we are approximating
# convention: an "M" as a prefix of the names of "exact" variables
```

```
My = x * x;
Ms1 = a2 + a4 * My;
Ms2 = Ms1 * My;
Ms = 1 + Ms2;
epsilon = (Ms-s);

# description of what we want to prove

{
# input hypothesis
|x| <= 1.23e-02

->
# goal to prove
|epsilon| in ?
/\ |s2| <= 1

# first line of goal: bound we wish to obtain
# second line: necessary to allow one to use Fast2Sum
}
```

When running this file with Gappa, we obtain

```
Results:
  |epsilon| in [0, 94384511554069319b-122 {1.77518e-20, 2^(-65.6106)}]
```

Concerning the first goal `|epsilon| in ?`, Gappa found the same error bound as the one we have computed in the previous section (which is not surprising: it probably uses the same method). There is no answer to our second goal `|s2|<= 1` which, in Gappa's syntax, just means that the answer was `true`.

For more complex programs, the way we have written the previous Gappa file is dangerous. We wanted to have an estimate of the error committed when evaluating $1+a_2x^2+a_4x^4$ using Algorithm 10. Imagine we have committed an error in the description of the program (hence quite possibly in the program itself), and that we have written

```
s1beforernd = a2 + a4*x;
```

instead of

```
s1beforernd = a2 + a4*y;
```

since the part of the Gappa file that describes the exact value was just obtained by directly rewriting, without roundings, the description of the program, we would very likely have also written

```
Ms1 = a2 + a4 * x;
```

instead of

```
Ms1 = a2 + a4 * My;
```

so that Gappa would have concluded that the computation is very accurate, although we do not at all compute what we wished to compute! The solution to that problem is to have a very simple description of the exact value, as close as possible to the mathematical definition and as independent as possible from the algorithm being used. We could for instance replace the four lines that describe the exact value by

```
Ms = 1 + a2*x*x + a4*x*x*x*x;
```

Unfortunately, if we just do that, we obtain a poor error bound. Gappa returns

```
Results:
  |epsilon| in [0, 174426593954067b-60 {0.000151291, 2^(-12.6904)}]
```

The solution is to give a hint to Gappa, i.e., to explain how the mathematical definition and the algorithm are related. This is done very simply, just by adding the line

```
Ms -> 1 + (a2 + a4*(x*x))*(x*x);
```

Gappa will try to check that the expressions `1 + a2*x*x + a4*x*x*x*x` and `1 + (a2 + a4*(x*x))*(x*x)` are equivalent, warn us if it does not succeed, and use the hint to compute a much better error bound, very close to the first one:

```
Results:
  |epsilon| in [0, 9439165181057033lb-122 {1.77531e-20, 2^(-65.6105)}]
```

Hence, our final Gappa file is as follows.

```
@RN = float<ieee_64,ne>;
# defines RN as round to the nearest in binary64 arithmetic

x = RN(xx);
a2 = -2251799813622611b-52;
a4 = 1501189276987675b-55;

# description of the program

y RN = x * x;
# now, we describe the action of the FMA
s1beforernd = a2 + a4*y;
s1 = RN(s1beforernd);
s2 RN = s1*y;
s = 1 + s2; # no rounding: Fast2Sum is an exact transformation

# description of the exact value we are approximating
# convention: an "M" as a prefix of the names of "exact" variables

Ms = 1 + a2*x*x + a4*x*x*x*x;
epsilon = (Ms-s);

# description of what we want to prove

{
# input hypothesis
|x| <= 1.23e-02

->
# goal to prove
|epsilon| in ?
/\ |s2| <= 1

# first line of goal: bound we wish to obtain
# second line: necessary to allow use of Fast2Sum

}

# Now some hints to help Gappa

Ms -> 1 + (a2 + a4*(x*x))*(x*x);
```

As we can see, compared to our approach of the previous section, Gappa frees us from long and error-prone calculations. Furthermore, once the initial Gappa input file is written, small modifications

allow one to easily explore variants of the evaluation scheme. The most important issue, however, is that if called with option `-Bcoq`, Gappa generates a formal proof of the returned result. That proof can then be verified by the Coq proof checker.[12]

2.2.5 Maple Programs that Compute binary32 and binary64 Approximations

The following Maple programs implement the round-to-nearest-even rounding functions in binary32/single-precision and binary64/double-precision. They compute the binary32 and binary64 floating-point numbers that are closest to t for any real number t (and they use the "round-to-nearest ties to even" tie-breaking rule).

RN function, binary32 arithmetic

```
nearest_binary32 := proc(xx)
local x, sign, logabsx, exponent, mantissa, infmantissa, powermin,
expmin, powermax, expmax, powermiddle, expmiddle;
Digits := 100;
x := evalf(xx);
if (x=0) then sign, exponent, mantissa := 1, -126, 0
else
  if (x < 0) then sign := -1
  else sign := 1
  fi:
  x := abs(x);
  if x >=  2^(127)*(2-2^(-24)) then mantissa := infinity; exponent := 127
  else if x <= 2^(-150) then mantissa := 0; exponent := -126
      else
         if x <= 2^(-126) then exponent := -126
         else
# x is between 2^(-126) and 2^(128)
         powermin := 2^(-126); expmin := -126;
         powermax := 2^128; expmax := 128;
         while (expmax-expmin > 1) do
            expmiddle := round((expmax+expmin)/2);
            powermiddle := 2^expmiddle;
            if x >= powermiddle then
                powermin := powermiddle;
                expmin := expmiddle
            else
                powermax := powermiddle;
                expmax := expmiddle
            fi
          od;
# now, expmax - expmin = 1
# and powermin <= x < powermax
# powermin = 2^expmin
# and powermax = 2^expmax
# so expmin is the exponent of x
         exponent := expmin;
         fi;
         infmantissa := x*2^(23-exponent);
         if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
            else
               mantissa := floor(infmantissa);
               if type(mantissa,odd) then mantissa := mantissa+1 fi
```

[12] Coq can be downloaded at https://coq.inria.fr.

```
            fi;
         mantissa := mantissa*2^(-23);
      fi;
  fi;
fi;
sign*mantissa*2^exponent;
end:
```

RN function, binary64 arithmetic

```
nearest_binary64 := proc(xx)
local x, sign, logabsx, exponent, mantissa, infmantissa,
powermin, expmin, powermax,expmax, powermiddle, expmiddle;
Digits := 100;
x := evalf(xx);
if (x=0) then sign, exponent, mantissa := 1, -1022, 0
else
  if (x < 0) then sign := -1
  else sign := 1
  fi:
  x := abs(x);
  if x >=  2^(1023)*(2-2^(-53)) then mantissa := infinity; exponent := 1023
  else if x <= 2^(-1075) then mantissa := 0; exponent := -1022
      else
         if x <= 2^(-1022) then exponent := -1022
         else
# x is between 2^(-1022) and 2^(1024)
         powermin := 2^(-1022); expmin := -1022;
         powermax := 2^1024; expmax := 1024;
         while (expmax-expmin > 1) do
            expmiddle := round((expmax+expmin)/2);
            powermiddle := 2^expmiddle;
            if x >= powermiddle then
                powermin := powermiddle;
                expmin := expmiddle
            else
                powermax := powermiddle;
                expmax := expmiddle
            fi
          od;
# now, expmax - expmin = 1
# and powermin <= x < powermax
# powermin = 2^expmin
# and powermax = 2^expmax
# so expmin is the exponent of x
         exponent := expmin;
         fi;
         infmantissa := x*2^(52-exponent);
         if frac(infmantissa) <> 0.5 then mantissa := round(infmantissa)
            else
               mantissa := floor(infmantissa);
               if type(mantissa,odd) then mantissa := mantissa+1 fi
            fi;
         mantissa := mantissa*2^(-52);
      fi;
  fi;
fi;
sign*mantissa*2^exponent;
end:
```

The following programs evaluates ulp (t) for any real number t, in binary32 and binary64 floating-point arithmetic.

ULP function, binary32 arithmetic

```
ulp_in_binary_32 := proc(t)
 local x, res, expmin, expmax, expmiddle;
 x := abs(t);
 if x < 2^(-125) then res := 2^(-149)
    else if x > (1-2^(-24))*2^(128) then res := 2^104
    else
      expmin := -125; expmax := 128;
 # x is between 2^expmin and 2^expmax
      while (expmax-expmin > 1) do
        expmiddle := round((expmax+expmin)/2);
        if x >= 2^expmiddle then
              expmin := expmiddle
        else  expmax := expmiddle
        fi;
       od;
 # now, expmax - expmin = 1
 # and 2^expmin <= x < 2^expmax
        res := 2^(expmin-23)
fi;
fi; res;
end:
```

ULP function, binary64 arithmetic

```
ulp_in_binary_64 := proc(t)
 local x, res, expmin, expmax, expmiddle;
 x := abs(t);
 if x < 2^(-1021) then res := 2^(-1074)
    else if x > (1-2^(-53))*2^(1024) then res := 2^971
    else
      expmin := -1021; expmax := 1024;
 # x is between 2^expmin and 2^expmax
      while (expmax-expmin > 1) do
        expmiddle := round((expmax+expmin)/2);
        if x >= 2^expmiddle then
              expmin := expmiddle
        else  expmax := expmiddle
        fi;
       od;
 # now, expmax - expmin = 1
 # and 2^expmin <= x < 2^expmax
        res := 2^(expmin-52)
fi;
fi:
res;
end:
```

2.2.6 The Future of Floating-Point Arithmetic

Floating-point arithmetic, as it is known nowadays, results from a compromise between several requirements, in terms of range, accuracy, ease of use, ease of implementation, ease of verification/certification,

speed, memory consumption…. As technology evolves, many parameters involved in that compromise change with time. A simple example is the ratio between the delay of a memory access and the delay of an arithmetic operation. This ratio has considerably increased during the last years. Ultimately, this evolution will almost certainly lead to changes in the way we represent and manipulate real numbers on computers. However, it is difficult to forecast the magnitude of these changes: will we use slightly modified versions of our current floating-point systems, or will we use very different number systems? Over the years, various alternatives to conventional floating-point arithmetic have been suggested: tapered floating-point arithmetic [13, 350], level index arithmetic [91, 369], logarithmic number systems [272, 441], slash number systems [344], etc. Recently, Gustafson [211] suggested an interesting variant of tapered floating-point arithmetic, called the Unum number system, with an "exact" bit added to the representation of the numbers, and a subtle interval interpretation of the nonexact representations.

2.3 Redundant Number Systems

In general, when we represent numbers in radix r, we use the digits 0, 1, 2, … $r-1$. And yet, sometimes, number systems using a different set of digits naturally arise. In 1840, Cauchy suggested the use of digits -5 to $+5$ in radix 10 to simplify multiplications [73]. Booth recoding [47] (a technique sometimes used by multiplier designers) generates numbers represented in radix 2, with digits -1, 0, and $+1$. Digit-recurrence algorithms for division and square root [179, 399] also generate results in a "signed-digit" representation.

Some of these exotic number systems allow carry-free addition. This is what we are going to investigate in this section.

First, assume that we want to compute the sum $s = s_n s_{n-1} s_{n-2} \dots s_0$ of two integers $x = x_{n-1} x_{n-2} \dots x_0$ and $y = y_{n-1} y_{n-2} \dots y_0$ represented in the conventional binary number system. By examining the well-known equation that describes the addition process ("$\vee$" is the boolean "or" and "$\oplus$" is the "exclusive or"):

$$\begin{aligned} c_0 &= 0 \\ s_i &= x_i \oplus y_i \oplus c_i \\ c_{i+1} &= x_i y_i \vee x_i c_i \vee y_i c_i \end{aligned} \tag{2.9}$$

we see that there is a dependency relation between c_i, the *incoming carry* at position i, and c_{i+1}. This does not mean that the addition process is intrinsically sequential, and that the sum of two numbers is computed in a time that grows linearly with the size of the operands: the addition algorithms and architectures proposed in the literature [180, 191, 277, 370, 378] and implemented in current microprocessors are much faster than a straightforward, purely sequential, implementation of (2.9). Nevertheless, the dependency relation between the carries makes a fully parallel addition impossible in the conventional number systems.

2.3.1 Signed-Digit Number Systems

In 1961, Avizienis [12] studied different number systems called *signed-digit* number systems. Let us assume that we use radix r. In a signed-digit number system, the numbers are no longer represented using digits between 0 and $r-1$, but with digits between $-a$ and a, where $a \leq r-1$. Every number is representable in such a system, if $2a \geq r-1$. For instance, in radix 10 with digits between -5 and

Figure 2.3 *Computation of* $1\overline{5}31\overline{2}0 + 1\overline{1}2\overline{6}1\overline{6}$ *using Avizienis' algorithm in radix* $r = 10$ *with* $a = 6$.

x_i	1	$\overline{5}$	3	1	$\overline{2}$	0
y_i	1	$\overline{1}$	$\overline{2}$	6	1	$\overline{6}$
$x_i + y_i$	2	-6	1	7	-1	-6
t_{i+1}	0	-1	0	1	0	-1
w_i	2	4	1	-3	-1	4
s_i	1	4	2	$\overline{3}$	$\overline{2}$	4

$+5$, every number is representable. The number 15725 can be represented by the digit chain $2\overline{4}325$ (we use $\overline{4}$ to represent the digit -4); i.e., $15725 = 2 \times 10^4 + (-4) \times 10^3 + (-3) \times 10^2 + 2 \times 10^1 + 5$.

The same number can also be represented by the digit chain $2\overline{4}3\overline{5}5$. If $2a \geq r$, then some numbers have several possible representations, which means that the number system is *redundant*. As shown later, this is an important property.

Avizienis also proposed addition algorithms for these number systems. Algorithm 11 performs the addition of two n-digit numbers $x = x_{n-1}x_{n-2}\cdots x_0$ and $y = y_{n-1}y_{n-2}\cdots y_0$ represented in radix r with digits between $-a$ and a, where $a \leq r - 1$ and[13] $2a \geq r + 1$.

Algorithm 11 Avizienis' algorithm

Input : $x = x_{n-1}x_{n-2}\cdots x_0$ and $y = y_{n-1}y_{n-2}\cdots y_0$
Output : $s = s_n s_{n-1}s_{n-2}\cdots s_0 = x + y$

1. in parallel, for $i = 0, \ldots, n-1$, compute t_{i+1} (carry) and w_i (intermediate sum) satisfying:

$$\begin{cases} t_{i+1} = \begin{cases} +1 \text{ if } x_i + y_i \geq a \\ \ \ 0 \text{ if } -a+1 \leq x_i + y_i \leq a-1 \\ -1 \text{ if } x_i + y_i \leq -a \end{cases} \\ w_i \ \ = x_i + y_i - r \times t_{i+1}. \end{cases} \tag{2.10}$$

2. in parallel, for $i = 0, \ldots, n$, compute $s_i = w_i + t_i$, with $w_n = t_0 = 0$.

By examining the algorithm, we can see that the carry t_{i+1} does not depend on t_i. There is no longer any carry propagation: all digits of the result can be generated simultaneously. The conditions "$2a \geq r+1$" and "$a \leq r-1$" cannot be simultaneously satisfied in radix 2. Nevertheless, it is possible to perform parallel, carry-free additions in radix 2 with digits equal to -1, 0, or 1, by using another algorithm, also due to Avizienis (or by using the *borrow-save adder* presented in the following).

Figure 2.3 presents an example of the execution of Avizienis' algorithm in the case $r = 10, a = 6$.

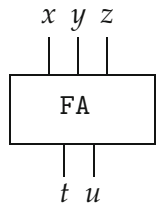
Redundant number systems are used in many instances: recoding of multipliers, quotients in division and division-like operations, online arithmetic [182], etc. Redundant additions are commonly used within arithmetic operators such as multipliers and dividers (the input and output data of such operators are represented in a nonredundant number system, but the internal calculations are performed in a redundant number system). For instance, most multipliers use (at least implicitly) the carry-save number system, whereas digit-recurrence dividers actually use two different number systems: the partial remainders are represented, in general, in carry-save, and the quotient digits are represented in

[13] This condition is stronger than the condition $2a \geq r - 1$ that is required to represent every number.

a signed-digit number system of radix 2^k, where k is a small integer [179]. The reader interested in redundant number systems can find useful information in [12, 180, 375, 376, 377, 382].

2.3.2 The Carry-Save and Borrow-Save Number Systems

Now let us focus on the particular case of radix 2. In this radix, the two common redundant number systems are the *carry-save* (CS) number system, and the signed-digit number system. In the carry-save number system, numbers are represented with digits 0, 1, and 2, and each digit d is represented by two bits $d^{(1)}$ and $d^{(2)}$ whose sum equals d. In the signed-digit number system, numbers are represented with digits -1, 0, and 1. In that system, we can represent the digits with the *borrow-save* (BS) encoding, also called (p, n) encoding [375]: each digit d is represented by two bits d^+ and d^- such that $d^+ - d^- = d$ (different encodings of the digits also lead to fast and simple arithmetic operators [88, 443]). Those two number systems allow very fast additions and subtractions. The *carry-save adder* (see, for instance, [277]) is a very well-known structure used for adding a number represented in the carry-save system and a number represented in the conventional binary system. It consists of a row of full-adder cells, where a full-adder cell computes two bits t and u, from three bits x, y, and z, such that $2t + u$ equals $x + y + z$ (see Figure 2.4). A carry-save adder is presented in Figure 2.5.



$$u = x \oplus y \oplus z$$
$$t = xy + xz + yz$$

Figure 2.4 *A full-adder (FA) cell. From three bits x, y, and z, it computes two bits t and u such that $x + y + z = 2t + u$.*

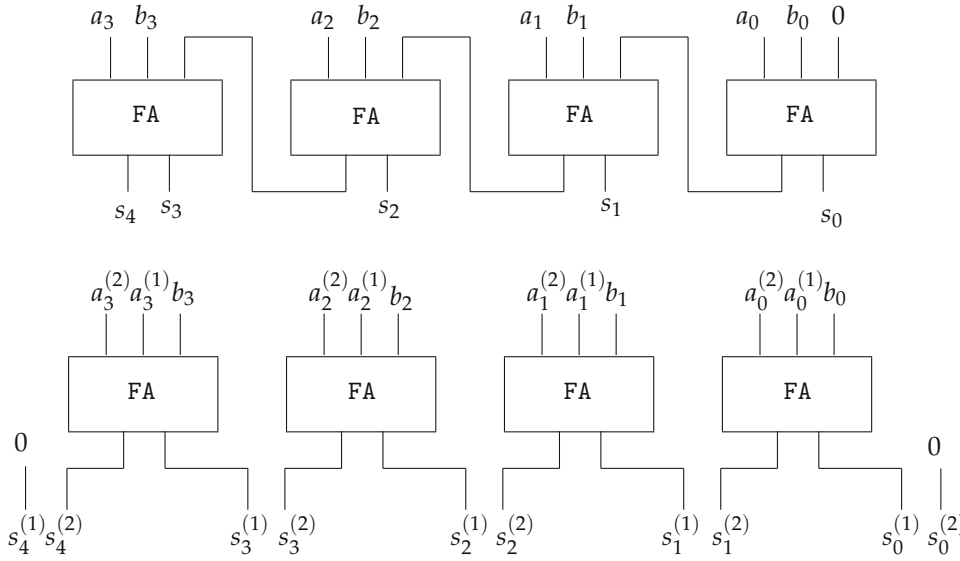


Figure 2.5 *A carry-save adder (bottom), compared to a carry-propagate adder (top).*

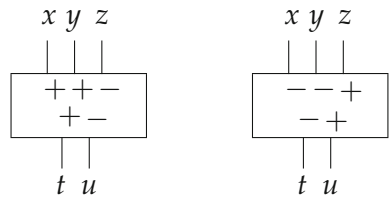


Figure 2.6 *A PPM cell. From three bits x, y, and z, it computes two bits t and u such that $x + y - z = 2t - u$.*

An adder structure for the borrow-save number system can easily be built using elementary cells slightly different from the FA cell. Algorithm 12 adds two BS numbers.

Algorithm 12 Borrow-Save addition

- input: two BS numbers $a = a_{n-1}a_{n-2}\cdots a_0$ and $b = b_{n-1}b_{n-2}\cdots b_0$, where the digits a_i and b_i belong to $\{-1, 0, 1\}$, each digit d being represented by two bits d^+ and d^- such that $d^+ - d^- = d$.
- output: a BS number $s = s_n s_{n-1}\cdots s_0$ satisfying $s = a + b$.

For each $i = 0, \ldots, n-1$, compute two bits c_{i+1}^+ and c_i^- such that $2c_{i+1}^+ - c_i^- = a_i^+ + b_i^+ - a_i^-$;
For each $i = 0, \ldots, n-1$, compute s_{i+1}^- and s_i^+ such that $2s_{i+1}^- - s_i^+ = c_i^- + b_i^- - c_i^+$ (with $c_0^+ = c_n^- = 0$, and $s_n^+ = c_n^+$).

Both steps of this algorithm require the same elementary computation: from three bits x, y, and z we must find two bits t and u such that $2t - u = x + y - z$. This can be done using a *PPM cell* ("PPM" stands for "Plus Plus Minus"), depicted in Figure 2.6, which is very similar to the FA cell previously described. Using PPM cells, one can easily derive the borrow-save adder of Figure 2.7 from the algorithm. It is possible to add a number represented in the borrow-save system and a number represented in the conventional, nonredundant, binary system by using only one row of PPM cells.[14] This is described in Figure 2.8. More details on borrow-save based arithmetic operators can be found in [24].

2.3.3 Canonical Recoding

Multiplying a given number a by a binary number $b = b_{n-1}b_{n-2}\cdots b_0 = \sum_{i=0}^{n-1} b_i 2^i$ reduces to computing

$$\sum_{i=0}^{n-1} b_i \cdot (a2^i),$$

i.e., we need to add as many shifted copies of a as there are nonzero values b_i. Very soon (first for accelerating multiplications when they were performed in software, using additions and shifts, and later on for accelerating hardwired multiplication and reducing the area of multipliers), authors tried to *recode* the operand b in order to reduce its number of nonzero digits. With the conventional binary representation (for which the only values allowed for b_i are 0 and 1), we have only one possible

[14]The carry-save and borrow-save systems are roughly equivalent: everything that is computable using one of these systems is computable at approximately the same cost as with the other one.

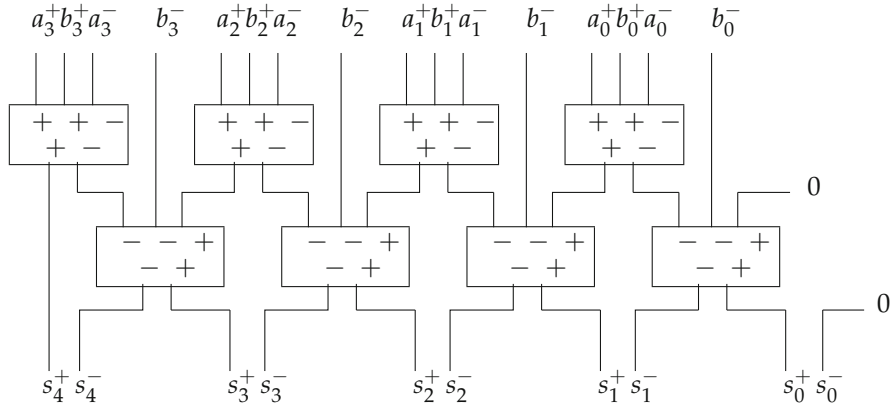


Figure 2.7 *A borrow-save adder.*

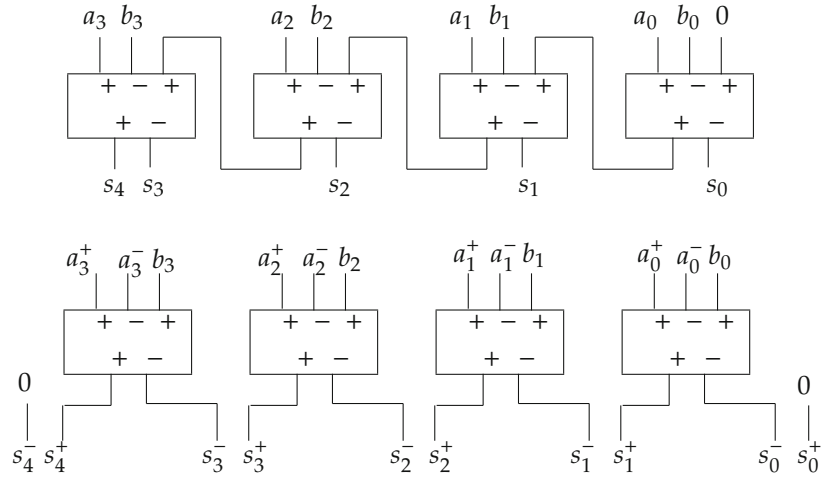


Figure 2.8 *A structure for adding a borrow-save number and a nonredundant number* (*bottom*), *compared to a carry-propagate subtractor* (*top*).

representation for each integer b, so attempting to reduce the number of nonzero digits makes no sense, but as we have seen before, if we allow digits from the larger digit set $\{-1, 0, 1\}$, we obtain a *redundant* number system: numbers have several possible representations, so that it makes sense to try to minimize the number of nonzero digits. In our initial multiplication problem, when $b_i = -1$, the number $a2^i$ is subtracted, which can be done with approximately the same delay and/or silicon area as an addition. For instance (still using the symbol $\overline{1}$ for representing the digit "-1"), the binary number

$$11101001111$$

can be "recoded"

$$100\overline{1}01010001\overline{1},$$

and it can be shown (it is a consequence of Theorem 6, below) that the number of nonzero digits of this last representation is minimal. Booth [47] first suggested to recode the initial binary chain by replacing all sub-strings of the form

$$0\underbrace{1\cdots 111}_{k \text{ ones}}$$

by

$$1\underbrace{0\cdots 00}_{k-1 \text{ zeros}}\overline{1}.$$

More formally, assuming that the initial n-bit binary chain is $d_{n-1}d_{n-2}\cdots d_0$, the Booth-recoded, $n+1$-digit chain, $f_n f_{n-1} f_{n-2}\cdots f_0$ is obtained using the set of rules given in Table 2.4.

Unfortunately, that set of rules does not always generate a digit string with a minimal number of nonzero digits. Indeed, the "recoded" digit chain may even have more nonzero digits than the initial one. Just consider the input chain

$$10101010101,$$

whose Booth recoding is

$$1\overline{1}1\overline{1}1\overline{1}1\overline{1}1\overline{1}.$$

Several authors suggested different recodings that are guaranteed to have less than $\lceil n/2 \rceil$ nonzero digits (where n is the length of the original binary chain). In his seminal paper [393], Reitwiesner suggested a recoding algorithm that generates digit chains that are "minimal," i.e., they have the smallest possible number of nonzero digits. Such digit chains are called *Canonical recodings*. Reitwiesner's algorithm consists in performing the transformation presented in Table 2.5. Looking at the table, one easily checks that we always have

$$2c_{i+1} + f_i = c_i + d_i,$$

from which we immediately deduce that

$$\sum_{i=0}^{n} f_i 2^i = \sum_{i=0}^{n-1} d_i 2^i,$$

i.e., the algorithm effectively generates a digit string that represents the input number.

Looking at the table, we can immediately find the following basic property of the digit strings generated by Reitwiesner's algorithm (which explains why the generated digit string is sometimes called *nonadjacent form* [208]).

Table 2.4 *The set of rules that generate the Booth recoding $f_n f_{n-1} f_{n-2}\cdots f_0$ of a binary number $d_{n-1}d_{n-2}\cdots d_0$ (with $d_i \in \{0, 1\}$ and $f_i \in \{-1, 0, 1\}$). By convention $d_n = d_{-1} = 0$.*

d_n	d_{n-1}	f_n
0	0	0
0	1	1
1	0	$\overline{1}$
1	1	0

Table 2.5 *Reitwiesner's algorithm: the set of rules that generate the canonical recoding $f_n f_{n-1} f_{n-2} \cdots f_0$ of a binary number $d_{n-1} d_{n-2} \cdots d_0$ (with $d_i \in \{0, 1\}$ and $f_i \in \{-1, 0, 1\}$). The process is initialized by setting $c_0 = 0$, and by convention $d_n = 0$.*

c_i	d_{i+1}	d_i	f_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	$\overline{1}$	1
1	0	0	1	0
1	0	1	0	1
1	1	0	$\overline{1}$	1
1	1	1	0	1

Property 4 (The nonzero digits of a digit string generated by Reitwiesner's algorithm are *nonadjacent.*) *If $f_n f_{n-1} \cdots f_0$ is the recoding of the binary string $d_{n-1} \cdots d_0$ deduced from the set of rules presented in Table 2.5 then for any i, $f_i \cdot f_{i+1} = 0$.*

A first consequence of this is that the canonical recoding of an n-bit binary string has at most $\lceil n/2 \rceil$ nonzero digits. A second consequence, due to Theorem 6 below, is that the digit strings generated by Reitwiesner's algorithm are minimal: they have the smallest possible number of nonzero digits.

Theorem 6 (Reitwiesner's theorem [393]: minimality of nonadjacent digit chains) *Assume an integer x is represented by a nonadjacent binary digit chain, i.e.,*

$$x = f_n f_{n-1} f_{n-2} \cdots f_0$$

with

$$\forall i, \begin{cases} f_i \in \{-1, 0, 1\}, \\ f_i \cdot f_{i+1} = 0 \end{cases}$$

Any binary representation of x with digits in $\{-1, 0, 1\}$ will contain as least as many nonzero digits as the digit chain $f_n f_{n-1} f_{n-2} \cdots f_0$.

Part I

Algorithms Based on Polynomial Approximation and/or Table Lookup, Multiple-Precision Evaluation of Functions

The Classical Theory of Polynomial or Rational Approximations

3

Using a finite number of additions, subtractions, multiplications, and comparisons, the only functions of one variable that one can compute are *piecewise polynomials*. If we add division to the set of available operations, the only functions one can compute are *piecewise rational functions*. Therefore, it is natural to try to approximate the elementary functions by polynomials or rational functions. The questions that immediately spring to mind are:

- How can we compute such polynomial or rational approximations?
- What is the best way (in terms of accuracy and/or speed) to evaluate a polynomial or a rational function?
- The final error will be the sum of two errors: the *approximation* error (i.e., the "distance" between the function being approximated and the polynomial or rational function), and the *evaluation* error due to the fact that the polynomial or rational functions are evaluated in finite precision floating-point arithmetic. Can we compute tight bounds on these errors?

We will try to address these questions in this chapter, and in Chapters 4 and 5. Throughout this chapter, we denote by $\mathcal{P}_n$ the set of the polynomials of degree less than or equal to n with real coefficients, and by $\mathcal{R}_{p,q}$ the set of the rational functions with real coefficients whose numerator and denominator have degrees less than or equal to p and q, respectively.

Let us focus first on the problem of building polynomial approximations. Of course, it is crucial to compute the coefficients of such approximations using a precision significantly higher than the "target precision" (i.e., the precision of the final result). We want to approximate a function f by an element p^* of $\mathcal{P}_n$ on an interval $[a, b]$. The methods presented in this chapter can be applied to any continuous function f (they are not limited to the elementary functions). Two kinds of approximations are considered here: the approximations that minimize the "average error," called *least squares approximations*, and the approximations that minimize the worst-case error, called *least maximum approximations*, or *minimax approximations*. In both cases, we want to minimize a "distance" $||p^* - f||$. For least squares approximations, that distance is

$$||p^* - f||_{2,[a,b]} = \sqrt{\int_a^b w(x)\,(f(x) - p^*(x))^2\,dx},$$

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_3

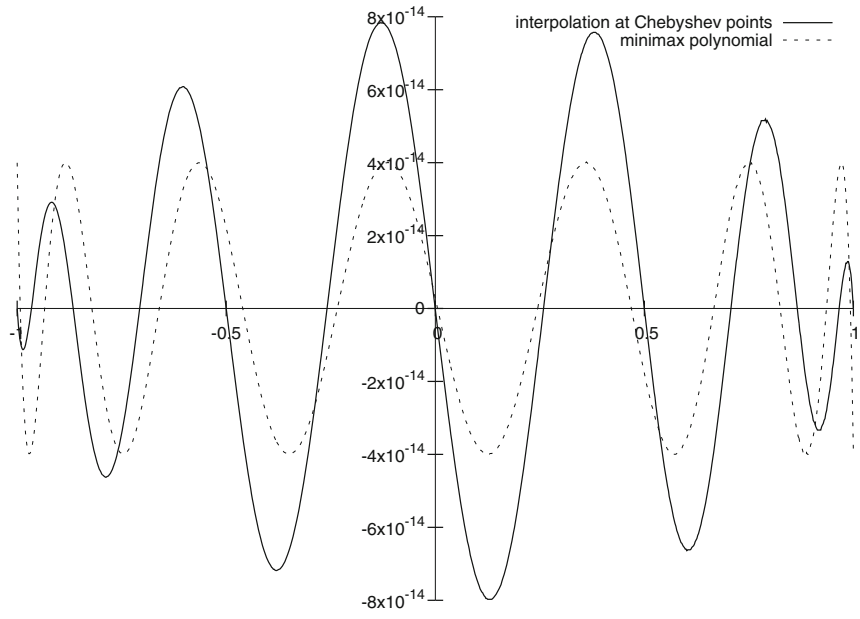


Figure 3.1 *Interpolation of* $\exp(x)$ *at the* 13 *Chebyshev points* $\cos(k\pi/12)$, $k = 0, 1, 2, \ldots, 12$, *compared to the degree*-12 *minimax approximation to* $\exp(x)$ *in* $[-1, 1]$. *The interpolation polynomial is nearly as good as the minimax polynomial.*

where w is a continuous, nonnegative, *weight function*, that can be used to select parts of $[a, b]$ where we want the approximation to be more accurate. For minimax approximations,[1] the distance is

$$||p^* - f||_{\infty,[a,b]} = \max_{a \le x \le b} w(x)|p^*(x) - f(x)|,$$

where $w = 1$ when we are interested in minimizing *absolute* errors, and $w(x) = 1/f(x)$ when we are interested in minimizing *relative* errors.

For the sake of simplicity, when there is no ambiguity on the interval of approximation, we will write $||p^* - f||_2$ instead of $||p^* - f||_{2,[a,b]}$, and $||p^* - f||_\infty$ instead of $||p^* - f||_{\infty,[a,b]}$.

3.1 What About Interpolation?

If one tries to approximate a continuous function by a polynomial, the first two ideas one has are, in general, to use Taylor series or to interpolate the function at some points. We will see later on in this chapter that in general Taylor series is not a good solution (at least for fixed-precision implementations: when multiple precision arithmetic is at stake, this may be quite different). Interpolation at *cleverly chosen points* could be a sensible solution: interpolating a function at Chebyshev points gives a polynomial almost as good as the minimax (see Section 3.3) polynomial, this is illustrated by Figure 3.1. Chebyshev interpolation is a very interesting tool, with nice mathematical properties and

[1] This kind of approximation is sometimes called Chebyshev approximation. Throughout this book, *Chebyshev approximation* means *least squares approximation using Chebyshev polynomials*. Chebyshev worked on both kinds of approximation.

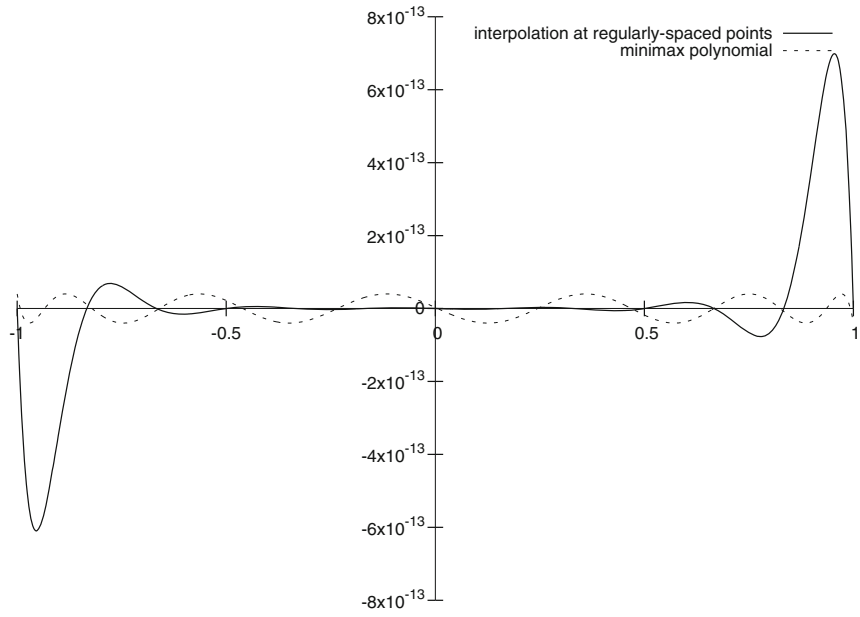


Figure 3.2 *Interpolation of* $\exp(x)$ *at* 13 *regularly spaced points, compared to the degree*-12 *minimax approximation to* $\exp(x)$ *in* $[-1, 1]$. *The minimax polynomial is much better than the interpolation polynomial, especially near both ends of the interval.*

many useful applications [459]. The reader interested by this topic can consult the excellent book by Trefethen [458] and try the Chebfun software system.[2] Trefethen notices in that book that for very large degrees, interpolating a function at Chebyshev points is much faster than computing its minimax approximation. This explains why for many (most?) numerical applications interpolation is preferable. However, the context of our work is particular: as function implementers, we never deal with degrees larger than a few tens. Furthermore, we can afford to spend time for building once for all finely tuned approximations that will be used millions of times: this leads us to favor approximation methods.

Notice that, although interpolation at Chebyshev points is a sensible option, interpolation at *regularly spaced* points is a catastrophic solution that will in general result in very poor approximations. This is illustrated by Figure 3.2.

3.2 Least Squares Polynomial Approximations

We are looking for a polynomial of degree $\leq n$,

$$p^*(x) = p_n^* x^n + p_{n-1}^* x^{n-1} + \cdots + p_1^* x + p_0^*$$

that satisfies

$$\int_a^b w(x) \left(f(x) - p^*(x)\right)^2 dx = \min_{p \in \mathcal{P}_n} \int_a^b w(x) \left(f(x) - p(x)\right)^2 dx. \tag{3.1}$$

Define $\langle f, g \rangle$ as

[2] Chebfun is available at www.chebfun.org.

$$\langle f, g\rangle = \int_a^b w(x) f(x) g(x) dx.$$

The approximation p^* can be computed as follows:

- build a sequence (T_m), $(m \leq n)$ of polynomials such that (T_m) is of degree m, and such that $\langle T_i, T_j\rangle = 0$ for $i \neq j$. Such polynomials are called *orthogonal polynomials*;
- compute the coefficients:
$$a_i = \frac{\langle f, T_i\rangle}{\langle T_i, T_i\rangle}; \tag{3.2}$$
- compute
$$p^* = \sum_{i=0}^{n} a_i T_i.$$

The proof is rather obvious and can be found in most textbooks on numerical analysis [202]. Some sequences of orthogonal polynomials, associated with simple weight functions w, are well known, so there is no need to compute them again. Let us now present some of them. More information on orthogonal polynomials can be found in [1, 203].

3.2.1 Legendre Polynomials

- weight function: $w(x) = 1$;
- interval $[a, b] = [-1, 1]$;
- definition:
$$\begin{cases} T_0(x) = 1 \\ T_1(x) = x \\ T_n(x) = \dfrac{2n-1}{n} x T_{n-1}(x) - \dfrac{n-1}{n} T_{n-2}(x); \end{cases}$$
- values of the scalar products:
$$\langle T_i, T_j\rangle = \begin{cases} 0 & \text{if } i \neq j \\ \dfrac{2}{2i+1} & \text{otherwise.} \end{cases}$$

3.2.2 Chebyshev Polynomials

- weight function: $w(x) = 1/\sqrt{1-x^2}$;
- interval $[a, b] = [-1, 1]$;
- definition:
$$\begin{cases} T_0(x) = 1 \\ T_1(x) = x \\ T_n(x) = 2x T_{n-1}(x) - T_{n-2}(x) = \cos\left(n \cos^{-1} x\right); \end{cases}$$
- values of the scalar products:

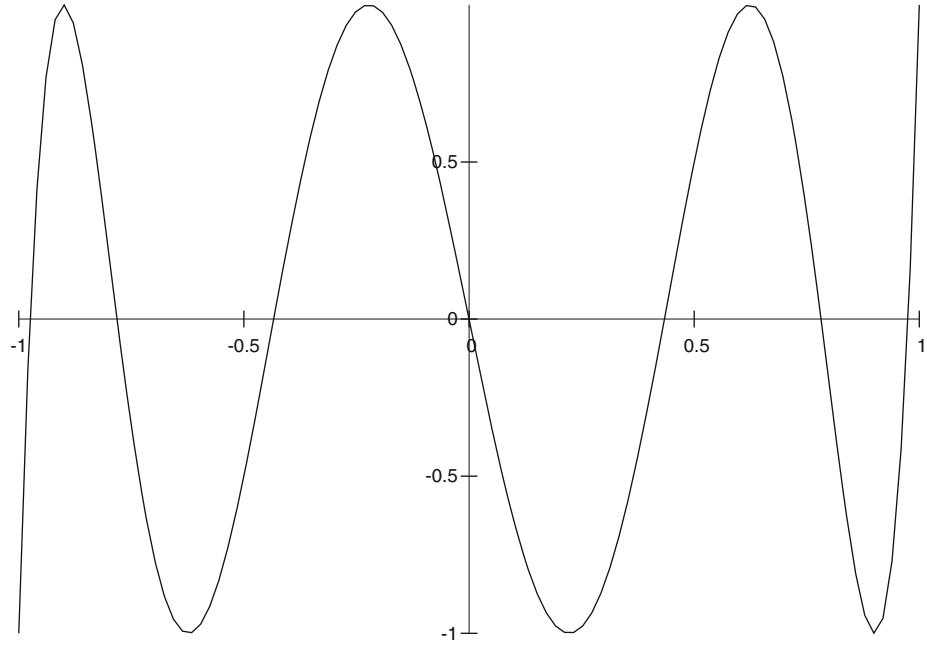


Figure 3.3 *Graph of the polynomial $T_7(x)$.*

$$\langle T_i, T_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ \pi & \text{if } i = j = 0 \\ \pi/2 & \text{otherwise.} \end{cases}$$

An example of a Chebyshev polynomial (T_7) is plotted in Figure 3.3.

Chebyshev polynomials play a central role in approximation theory. Among their many properties, the following three are frequently used. A much more detailed presentation of the Chebyshev polynomials can be found in [52, 398, 458].

Theorem 7 *For $n \geq 0$, we have*

$$T_n(x) = \frac{n}{2} \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{(n-k-1)!}{k!(n-2k)!} (2x)^{n-2k}.$$

Hence, the leading coefficient of T_n is 2^{n-1}. T_n has n real roots, all strictly between -1 and 1.

Theorem 8 *There are $n+1$ points $x_0, x_1, x_2, \ldots, x_n$ satisfying*

$$-1 = x_0 < x_1 < x_2 < \cdots < x_n = 1$$

such that

$$T_n(x_i) = (-1)^{n-i} \max_{x \in [-1,1]} |T_n(x)| \quad \forall i,\ i = 0, \ldots, n.$$

That is, the maximum absolute value of T_n is attained at the x_i's, and the sign of T_n alternates at these points.

Let us call a *monic* polynomial a polynomial whose leading coefficient is 1. We have,

Theorem 9 (Monic polynomials of smallest norm) *Let a, b be real numbers, with $a \leq b$. The monic degree-n polynomial P that minimizes*

$$\max_{x\in[a,b]} |P(x)|$$

is

$$\frac{(b-a)^n}{2^{2n-1}} T_n\left(\frac{2x-b-a}{b-a}\right).$$

3.2.3 Jacobi Polynomials

- weight function: $w(x) = (1-x)^\alpha (1+x)^\beta$ $(\alpha, \beta > 1)$;
- interval $[a, b] = [-1, 1]$;
- definition:

$$T_n(x) = \frac{1}{2^n} \sum_{m=0}^{n} \binom{n+\alpha}{m} \binom{n+\beta}{n-m} (x-1)^{n-m} (x+1)^m;$$

- values of the scalar products:

$$\langle T_i, T_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ h_i & \text{otherwise.} \end{cases}$$

with

$$h_i = \frac{2^{\alpha+\beta+1}}{2i+\alpha+\beta+1} \frac{\Gamma(i+\alpha+1)\Gamma(i+\beta+1)}{i!\Gamma(i+\alpha+\beta+1)}.$$

3.2.4 Laguerre Polynomials

- weight function: $w(x) = e^{-x}$;
- interval $[a, b] = [0, +\infty]$;
- definition:

$$T_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n}(x^n e^{-x});$$

- values of the scalar products:

$$\langle T_i, T_j \rangle = \begin{cases} 0 \text{ if } i \neq j \\ 1 \text{ otherwise.} \end{cases}$$

3.2.5 Using These Orthogonal Polynomials in Any Interval

Except for the Laguerre polynomials, for which $[a, b] = [0, +\infty]$, the orthogonal polynomials we have given are for the interval $[-1, 1]$. Getting an approximation for another interval $[a, b]$ is straightforward:

- for $u \in [-1, 1]$, define
$$g(u) = f\left(\frac{b-a}{2}u + \frac{a+b}{2}\right);$$
notice that $x = ((b-a)/2)u + ((a+b)/2) \in [a, b]$;
- compute a least squares polynomial approximation q^* to g in $[-1, 1]$;
- get the least squares approximation to f, say p^*, as
$$p^*(x) = q^*\left(\frac{2}{b-a}x - \frac{a+b}{b-a}\right).$$

3.3 Least Maximum Polynomial Approximations

As in the previous section, we want to approximate a continuous function f by a polynomial $p^* \in \mathcal{P}_n$ on a closed interval $[a, b]$. Let us assume the weight function $w(x)$ equals 1. In the following, $||f - p||_{\infty,[a,b]}$ (or $||f - p||_\infty$ for short when there is no ambiguity on the interval of approximation) denotes the *distance*:
$$||f - p||_{\infty,[a,b]} = \max_{a \le x \le b} |f(x) - p(x)|.$$

The norm $|| \cdot ||_\infty$ is often called *supremum norm*, or L^∞ norm. We look for a polynomial p^* that satisfies
$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_n} ||f - p||_\infty.$$

The polynomial p^* exists and is unique. It is called the *minimax* degree-n polynomial approximation to f on $[a, b]$. In 1885, Weierstrass proved the following theorem, which shows that a continuous function can be approximated as accurately as desired by a polynomial.

Theorem 10 (Weierstrass 1885) *Let f be a continuous function on $[a, b]$. For any $\epsilon > 0$ there exists a polynomial p such that $||p - f||_{\infty,[a,b]} \le \epsilon$.*

Another theorem, due to Chebyshev,[3] gives a characterization of the minimax approximations to a function.

Theorem 11 (Chebyshev) *p^* is the minimax degree-n approximation to f on $[a, b]$ if and only if there exist at least $n + 2$ values*
$$a \le x_0 < x_1 < x_2 < \cdots < x_{n+1} \le b$$
such that
$$p^*(x_i) - f(x_i) = (-1)^i \left[p^*(x_0) - f(x_0)\right] = \pm||f - p^*||_\infty.$$

See for instance [458] or [388] for a proof. The books by Cheney [81] and Phillips [385] are also good references. Theorem 11 is illustrated in Figures 3.4 and 3.5 for the case $n = 3$.

[3] According to Trefethen [458], that result was known by Chebyshev but the first proof was given by Kirchberger in his Ph.D. dissertation [273].

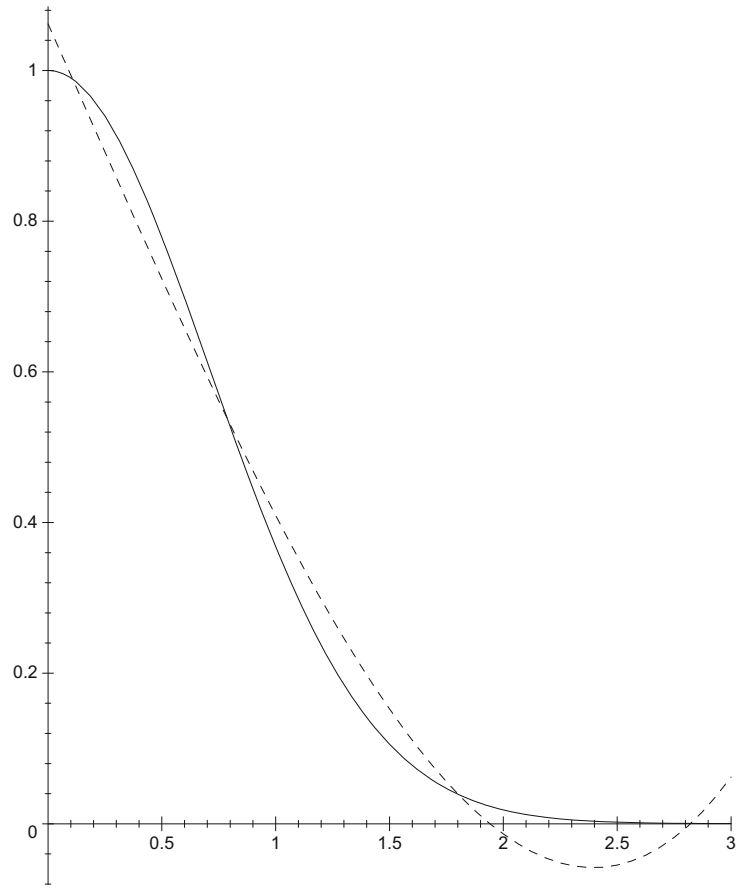


Figure 3.4 *The* $\exp(-x^2)$ *function and its degree-*3 *minimax approximation on the interval* $[0, 3]$ *(dashed line). There are five values where the maximum approximation error is reached with alternate signs.*

Chebyshev's theorem shows that if p^* is the minimax degree-n approximation to f, then the largest approximation error is reached at least $n + 2$ times, and that the sign of the error *alternates*. That "equioscillating" property allows us to directly find p^* in some particular cases, as we show in Section 3.4. It is used by an algorithm, due to Remez [225, 394] (see Section 3.6), that computes the minimax degree-n approximation to a continuous function iteratively. The reader can consult the seminal work by de La Vallée Poussin [140], Rice's book [396], and a survey by Fraser [196].

It is worth noticing that in some cases, $p^* - f$ may have more than $n + 2$ extrema. Figure 3.6 presents the minimax polynomial approximations of degrees 3 and 5 to the sine function in $[0, 4\pi]$. For instance, $\sin(x) - p_3^*(x)$ (where p_3^* is the degree-3 minimax approximation) has 6 extrema in $[0, 4\pi]$.

3.4 Some Examples

Example 1 (Approximations to e^x by Degree-2 Polynomials) Assume now that we want to compute a degree-2 polynomial approximation to the exponential function on the interval $[-1, 1]$. We use some of the methods previously presented, to compute and compare various approximations.

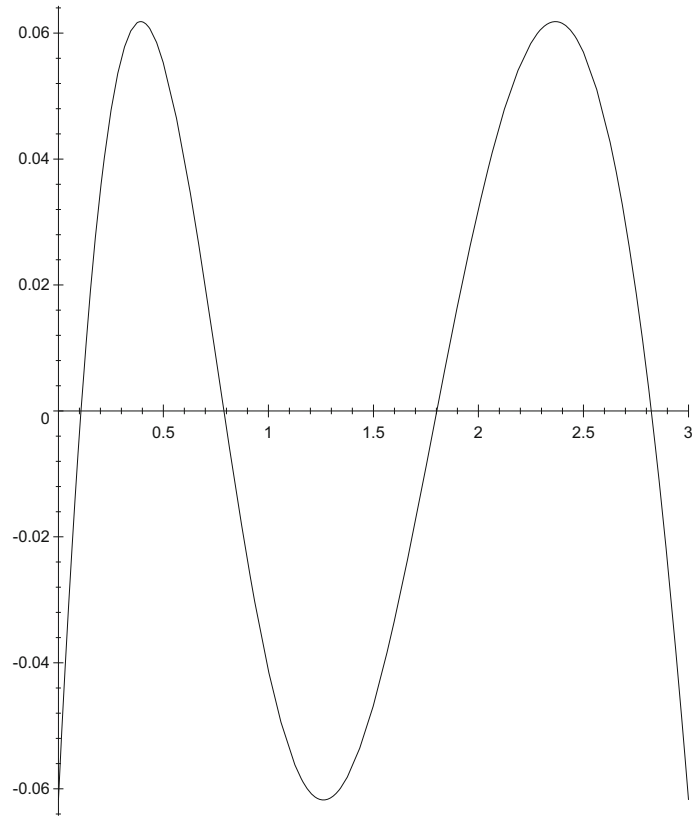


Figure 3.5 *The difference between the* $\exp(-x^2)$ *function and its degree-*3 *minimax approximation on the interval* $[0, 3]$.

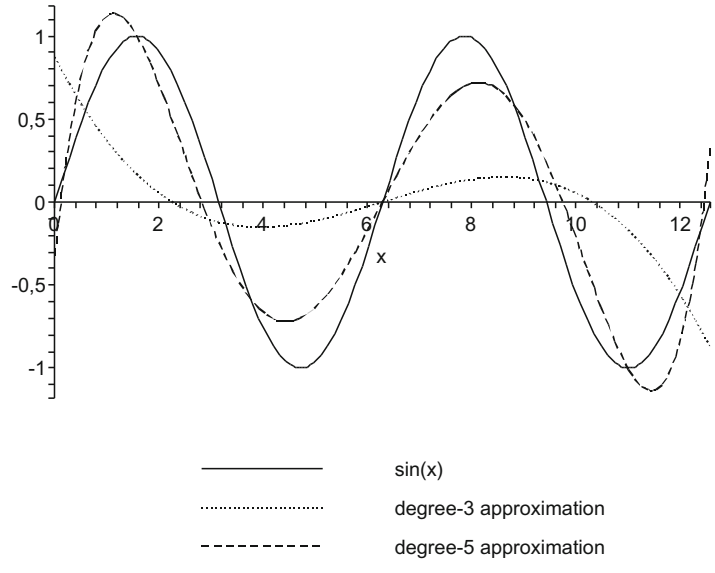


Figure 3.6 *The minimax polynomial approximations of degrees* 3 *and* 5 *to* $\sin(x)$ *in* $[0, 4\pi]$. *Notice that* $\sin(x) - p_3(x)$ *has* 6 *extrema. From Chebyshev's theorem, we know that it must have at least* 5 *extrema.*

Least squares approximation using Legendre polynomials

The first three Legendre polynomials are

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= \tfrac{3}{2}x^2 - \tfrac{1}{2}. \end{aligned}$$

The scalar product associated with Legendre approximation is

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)dx.$$

One easily gets

$$\begin{aligned} \langle e^x, T_0\rangle &= e - 1/e \\ \langle e^x, T_1\rangle &= 2/e \\ \langle e^x, T_2\rangle &= e - 7/e \\ \langle T_0, T_0\rangle &= 2 \\ \langle T_1, T_1\rangle &= 2/3 \\ \langle T_2, T_2\rangle &= 2/5. \end{aligned}$$

Therefore, the coefficients a_i of Eq. (3.2) are $a_0 = (1/2)\,(e - 1/e), a_1 = 3/e, a_2 = (5/2)\,(e - 7/e)$, and the polynomial $p^* = a_0T_0 + a_1T_1 + a_2T_2$ is equal to

$$\frac{15}{4}\left(e - \frac{7}{e}\right)x^2 + \frac{3}{e}x + \frac{33}{4e} - \frac{3e}{4} \simeq 0.5367215x^2 + 1.103683x + 0.9962940.$$

Least squares approximation using Chebyshev polynomials

The first three Chebyshev polynomials are:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1. \end{aligned}$$

The scalar product associated with Chebyshev approximation is

$$\langle f, g\rangle = \int_{-1}^{1} \frac{f(x)g(x)}{\sqrt{1 - x^2}}dx.$$

Using any numerical integration algorithm, one can get

$$\begin{aligned} \langle e^x, T_0\rangle &= 3.977463261\cdots \\ \langle e^x, T_1\rangle &= 1.775499689\cdots \\ \langle e^x, T_2\rangle &= 0.426463882\cdots. \end{aligned}$$

Therefore, since $\langle T_0, T_0\rangle = \pi$, and $\langle T_i, T_i\rangle = \pi/2$ for $i > 0$, the coefficients a_i of Eq. (3.2) are $a_0 = 1.266065878$, $a_1 = 1.130318208$, $a_2 = 0.2714953395$, and the polynomial $p^* = a_0T_0 + a_1T_1 + a_2T_2$ is approximately equal to

$$0.5429906776x^2 + 1.130318208x + 0.9945705392.$$

Minimax approximation

Assume that $p^*(x) = a_0 + a_1x + a_2x^2$ is the minimax approximation to e^x on $[-1, 1]$. From Theorem 11, there exist at least four values x_0, x_1, x_2, and x_3 where the maximum approximation error is reached with alternate signs. The convexity of the exponential function implies $x_0 = -1$ and $x_3 = +1$. Moreover, the derivative of $e^x - p^*(x)$ is equal to zero for $x = x_1$ and x_2. This gives

$$\begin{cases} a_0 - a_1 + a_2 - 1/e & = & \epsilon \\ a_0 + a_1x_1 + a_1x_1^2 - e^{x_1} & = & -\epsilon \\ a_0 + a_1x_2 + a_2x_2^2 - e^{x_2} & = & \epsilon \\ a_0 + a_1 + a_2 - e & = & -\epsilon \\ a_1 + 2a_2x_1 - e^{x_1} & = & 0 \\ a_1 + 2a_2x_2 - e^{x_2} & = & 0. \end{cases} \tag{3.3}$$

The solution of this nonlinear system of equations is

$$\begin{cases} a_0 = & 0.98903973\cdots \\ a_1 = & 1.13018381\cdots \\ a_2 = & 0.55404091\cdots \\ x_1 = & -0.43695806\cdots \\ x_2 = & 0.56005776\cdots \\ \epsilon = & 0.04501739\cdots \end{cases} \tag{3.4}$$

Therefore, the best minimax degree-2 polynomial approximation to e^x in $[-1, 1]$ is $0.98903973 + 1.13018381x + 0.55404091x^2$, and the largest approximation error is 0.045.

Table 3.1 presents the maximum errors obtained for the various polynomial approximations examined in this example, and the error obtained by approximating the exponential function by its degree-2 Taylor expansion at 0, namely,

$$e^x \approx 1 + x + \frac{x^2}{2}.$$

One can see that the Taylor expansion is much worse than the other approximations. This happens usually: Taylor expansions only give *local* (i.e., around one value) approximations, and should not in general be used for *global* (i.e., on an interval) approximations. The differences between the exponential function and its approximants are plotted in Figure 3.7: we see that Legendre approximation is the best "on average," that the minimax approximation is the best in the worst cases, and that Chebyshev approximation is very close to the minimax approximation.

Table 3.1 *Maximum absolute errors for various degree-2 polynomial approximations to* e^x *on* $[-1, 1]$.

	Taylor	Legendre	Chebyshev	Minimax
Max. error	0.218	0.081	0.050	0.045

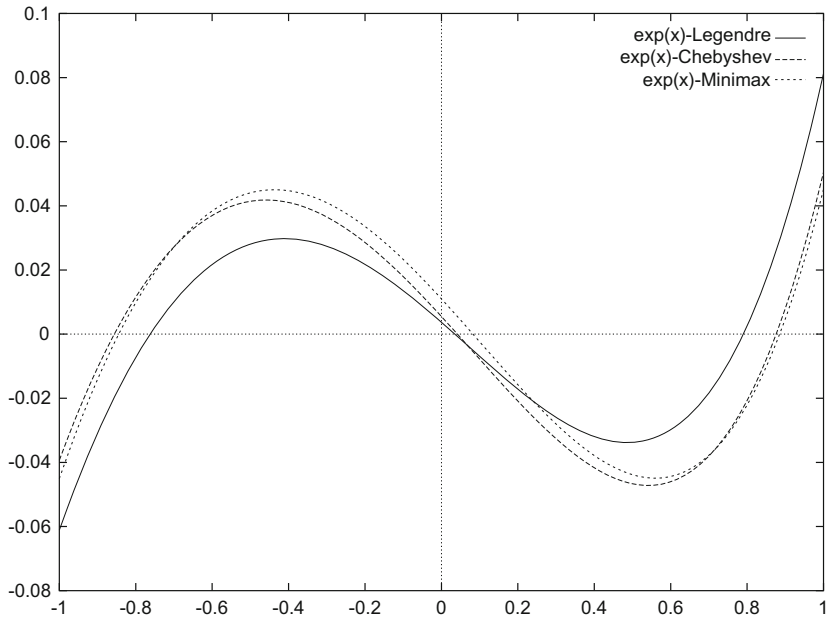


Figure 3.7 *Errors of various degree-2 approximations to e^x on $[-1, 1]$. Legendre approximation is better on average, and Chebyshev approximation is close to the minimax approximation.*

As shown in the previous example, Taylor expansions generally give poor polynomial approximations when the degree is sufficiently high, and should be avoided.[4] Let us consider another example. We wish to approximate the sine function in $[0, \pi/4]$ by a degree-11 polynomial. The error of the minimax approximation is 0.5×10^{-17}, and the maximum error of the Taylor expansion is 0.7×10^{-11}. In this example, the Taylor expansion is more than one million times less accurate.

It is sometimes believed that the minimax polynomial approximation to a function f is obtained by computing an expansion of f on the Chebyshev polynomials. This confusion is probably due not only to the fact that Chebyshev worked on both kinds of approximations, but also to the following property. As pointed out by Hart et al. [225], if the function being approximated is regular enough, then its Chebyshev approximation is very close to its minimax approximation (this is the case for the exponential function, see Figure 3.7). More precisely, if p_n^* is the minimax degree-n approximation to some continuous function f on $[-1, 1]$, and if p_n is its degree-n Chebyshev approximation, then (Trefethen [458]),

$$||f - p_n||_{\infty,[-1,1]} \leq \left(4 + \frac{4}{\pi}\log(n+1)\right) \cdot ||f - p_n^*||_{\infty,[-1,1]}.$$

If f is very regular, an even closer bound can be shown: Li [314] showed that when the function being approximated is an elementary function, the minimax approximation is at most one bit more accurate than the Chebyshev approximation.

We must notice, however, that for an irregular enough function, the Chebyshev approximation is not so close to the minimax approximation: this is illustrated by the next example.

[4] An exception is multiple precision computations (see Chapter 7), since it is not possible to precompute and store least squares or minimax approximations for all possible precisions.

Example 2 (Approximations to $|x|$ by Degree-2 Polynomials)

In the previous example, we tried to approximate a very regular function (the exponential) by a polynomial. We saw that even with polynomials of degree as small as 2, the approximations were quite good. Irregular functions are more difficult to approximate. Let us study the case of the approximation to $|x|$, between -1 and $+1$, by a degree-2 polynomial. By performing computations similar to those of the previous example, we get

- Legendre approximation:
$$\frac{15}{16}x^2 + \frac{3}{16} = 0.9375x^2 + 0.1875;$$
- Chebyshev approximation:
$$\frac{8}{3\pi}x^2 + \frac{2}{3\pi} \simeq 0.8488263x^2 + 0.21220659;$$
- Minimax approximation:
$$x^2 + \frac{1}{8}.$$

Those functions are plotted in Figure 3.8, and the worst-case errors are presented in Table 3.2.

Table 3.2 and Figure 3.8 show that in the case of the function $|x|$, the Chebyshev and the minimax approximations are quite different (the worst-case error is less for the Legendre approximation than for the Chebyshev approximation).

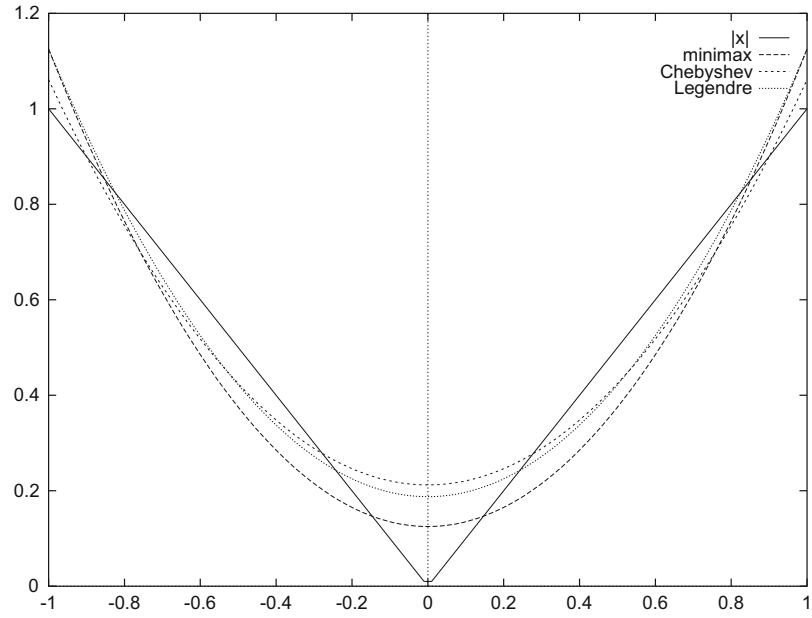


Figure 3.8 *Comparison of Legendre, Chebyshev, and minimax degree-2 approximations to* $|x|$.

Table 3.2 *Maximum absolute errors for various degree-2 polynomial approximations to* $|x|$ *on* $[-1, 1]$.

	Legendre	Chebyshev	Minimax
Max. error	0.1875	0.2122	0.125

Table 3.3 *Number of significant bits (obtained as* $-\log_2$(absolute error)*) of the minimax approximations to various functions on* $[0, 1]$ *by polynomials of degree 2 to 8. The accuracy of the approximation changes drastically with the function being approximated.*

Function\degree	2	3	4	5	6	7	8	9
$\sin(x)$	7.8	12.7	16.1	21.6	25.5	31.3	35.7	41.9
e^x	6.8	10.8	15.1	19.8	24.6	29.6	34.7	40.1
$\ln(1+x)$	8.2	11.1	14.0	16.8	19.6	22.3	25.0	27.7
$(x+1)^x$	6.3	8.5	11.9	14.4	18.1	20.0	22.7	25.1
$\arctan(x)$	8.7	9.8	13.2	15.5	17.2	21.2	22.3	24.5
$\tan(x)$	4.8	6.9	8.9	10.9	12.9	14.9	16.9	19.0
$\sqrt{x}$	3.9	4.4	4.8	5.2	5.4	5.6	5.8	6.0
$\arcsin(x)$	3.4	4.0	4.4	4.7	4.9	5.1	5.3	5.5

3.5 Speed of Convergence

We have seen in the previous sections that any continuous function can be approximated as closely as desired by a polynomial. Unfortunately, when the function is not regular enough, to reach a given approximation error, the degree of the required approximation polynomial may be quite large. A theorem due to Bernstein [225] shows that the convergence of the degree-n minimax approximations toward the function may be very slow. If we select a "speed of convergence" by choosing a decreasing sequence (ϵ_n) of positive real numbers such that $\epsilon_n \to 0$, there exists a continuous function f such that the approximation error of the minimax degree-n polynomial approximation to f is equal to ϵ_n; that is, the sequence of minimax polynomials converges to f with the "speed of convergence" that we have chosen.

Table 3.3 presents the speed of convergence of the polynomial approximations to some usual functions. One can see that the speed of convergence seems difficult to predict (there are, however, results that show that the speed depends on the function being analytic, or entire, or not [458]). Figure 3.9 plots the figures given in the table.

3.6 Remez's Algorithm

Since Remez's algorithm plays a central role in least maximum approximation theory, we give a brief presentation of it. We must warn the reader that, even if the outlines of the algorithm are reasonably simple, making sure that an implementation will always return a valid result is sometimes quite difficult [196]. An experienced user might prefer to write his/her own minimax approximation programs, to have a better control of the various parameters. A beginner or an occasional user will probably be well advised to use the polynomial approximation routines provided by softwares such as Sollya, Maple, or Mathematica.

For approximating a function f in the interval $[a, b]$, Remez's algorithm consists in iteratively building the set of points $x_0, x_1, \ldots, x_{n+1}$ of Theorem 11. We proceed as follows.

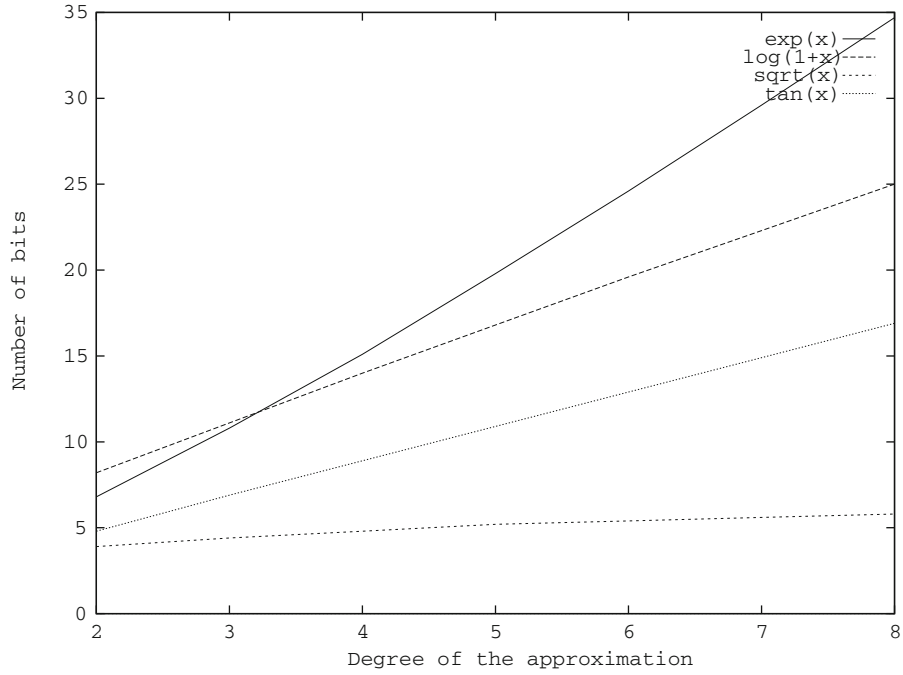


Figure 3.9 *Number of significant bits (obtained as* $-\log_2(error)$*) of the minimax polynomial approximations to various functions on* [0, 1].

1. We start from an initial set of points $x_0, x_1, \ldots, x_{n+1}$ in $[a, b]$.
2. We consider the linear system of equations

$$\begin{cases} p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_n x_0^n - f(x_0) = +\epsilon \\ p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_n x_1^n - f(x_1) = -\epsilon \\ p_0 + p_1 x_2 + p_2 x_2^2 + \cdots + p_n x_2^n - f(x_2) = +\epsilon \\ \cdots \qquad\qquad\qquad\qquad\qquad\qquad\quad \cdots \\ p_0 + p_1 x_{n+1} + \cdots + p_n x_{n+1}^n - f(x_{n+1}) = (-1)^{n+1}\epsilon. \end{cases} \tag{3.5}$$

   It is a system of $n + 2$ linear equations, with $n + 2$ unknowns: p_0, p_1, p_2, ..., p_n, and ϵ. That system is never singular (the corresponding matrix is a Vandermonde matrix [327]). Hence, it will have exactly one solution $(p_0, p_1, \ldots, p_n, \epsilon)$. Solving this system gives a polynomial $P(x) = p_0 + p_1 x + \cdots + p_n x^n$.
3. We now compute the set of points y_i in $[a, b]$ where $P - f$ has its extremes, and we start again (step 2), replacing the $x_i's$ by the y_i's.

It can be shown [196] that this is a convergent process, and that the speed of convergence is quadratic [461]. In general, starting from the initial set of points

$$x_i = \frac{a+b}{2} + \frac{(b-a)}{2} \cos\left(\frac{i\pi}{n+1}\right), 0 \le i \le n+1,$$

i.e., the points at which $|T_{n+1}((2x - b - a)/(b - a))| = 1$, where T_i is the Chebyshev polynomial of degree i, is advisable. This comes from the fact that minimax approximation and approximation using

Chebyshev polynomials are very close in most usual cases. The following Maple program, derived from one due to Paul Zimmermann, implements this algorithm. It is a "toy program" whose purpose is to help the reader to play with the algorithm. It will work reasonably well provided that we always find *exactly* $n+2$ points in $[a,b]$ where $P-f$ has its extremes, and that a and b are among these points. This will be the case in general.

First, this is a procedure that computes all roots of a given function g in the interval $[a,b]$, assuming that no interval of the form $[a+kh, a+(k+1)h]$, where $h=(b-a)/200$, contains more than one root.

```
AllRootsOf := proc(g,a,b);
# divides [a,b] into 200 subintervals
# and assumes each subinterval contains at most
# one root of function g
ListOfSol := [];
h := (b-a)/200;
for k from 0 to 199 do
  left := a+k*h;
  right := left+h;
  if evalf(g(left)*g(right)) <= 0 then
     sol := fsolve(g(x),x,left..right);
     ListOfSol := [op(ListOfSol),sol]
  end if;
end do;
ListOfSol
end;
```

Now, here is Remez's algorithm.

```
Remez := proc(f, x, n, a, b)
    P := add(p[i]*x^i, i = 0 .. n);
    pts := sort([seq(evalf(1/2*a + 1/2*b
        + 1/2*(b - a)*cos(Pi*i/(n + 1))),
        i = 0 .. n + 1)]);
# we initialize the set of points xi with the Chebyshev
# points
     ratio := 2;
Count := 1;     threshold := 1.000005;
     while ratio > threshold do
        sys := {seq(evalf(subs(x =
           op(i + 1, pts), P - f)) = (-1)^i*eps,
            i = 0 .. n + 1)};

printf("ITERATION NUMBER: %a\n",Count);
printf("Current list of points: %a\n",pts);
Count := Count+1;
printf("Linear system: %a\n",sys);
        sys := solve(sys, {eps, seq(p[i], i = 0 .. n)});
# we compute the polynomial associated with the list of
# points
        oldq := q;
        q := subs(sys, P);
        printf("Current polynomial: %a\n",q);
# we now compute the new list of points
# by looking for the extremes of q-f
derivative := unapply(diff(q-f,x),x);
pts := AllRootsOf(derivative,a,b);
no := nops(pts);
if no > n+2 then print("Too many extreme values,
```

```
    try larger degree")
    elif no = n then pts := [a,op(pts),b]
      elif no = n+1 then
        if abs((q-f)(a)) > abs((q-f)(b))
                      then pts := [a,op(pts)]
            else pts :=  [op(pts),b]
        end if
      elif no < n then print("Not enough oscillations")
end if;
lprint(pts);
  Emax := evalf(subs(x=pts[1],abs (q-f)));
        Emin := Emax;
        for i from 2 to (n+2) do
           Ecurr :=  evalf(subs(x=pts[i],abs (q-f)));
           if Ecurr > Emax then Emax := Ecurr
             elif Ecurr < Emin then Emin := Ecurr fi
           end do;
         ratio := Emax/Emin;
# We consider that we have found the Minimax polynomial
# (i.e., that the conditions of Chebyshev's
#                              theorem are met)
# when 1 < Emax/Emin < threshold
# threshold must be very slightly above 1
         printf("error: %a\n",Emax);
      end do;
    q
end proc;
```

To illustrate the behavior of Remez's algorithm, let us consider the computation, with the above given Maple program, of a degree-4 minimax approximation to $\sin(\exp(x))$ in $[0, 2]$.

We start from the following list of points: 0, 0.1909830057, 0.6909830062, 1.309016994, 1.809016994, 2, i.e., the points

$$1 + \cos\left(\frac{i\pi}{5}\right), i = 0, \ldots, 5,$$

that is, the points at which $|T_5(x-1)| = 1$.

The corresponding linear system is

$$\begin{cases} p_0 - 0.8414709848 & & & = \epsilon \\ p_0 + 0.1909830057 p_1 & + 0.03647450847 p_2 & + 0.00696601126 p_3 & \\ & + 0.00133038977 p_4 & - 0.9357708449 & = -\epsilon \\ p_0 + 0.6909830062 p_1 & + 0.4774575149 p_2 & + 0.3299150289 p_3 & \\ & + 0.2279656785 p_4 & - 0.9110882027 & = \epsilon \\ p_0 + 1.309016994 p_1 & + 1.713525491 p_2 & + 2.243033987 p_3 & \\ & + 2.936169607 p_4 & + 0.5319820928 & = -\epsilon \\ p_0 + 1.809016994 p_1 & + 3.272542485 p_2 & + 5.920084968 p_3 & \\ & + 10.70953431 p_4 & + 0.1777912944 & = \epsilon \\ p_0 + 2 p_1 & + 4 p_2 & + 8 p_3 & \\ & + 16 p_4 & - 0.8938549549 & = -\epsilon. \end{cases}$$

Solving this system gives the following polynomial:

$$P^{(1)}(x) = 0.7808077493 + 1.357210937x$$
$$-0.7996276765x^2 - 2.295982186x^3 + 1.189103547x^4.$$

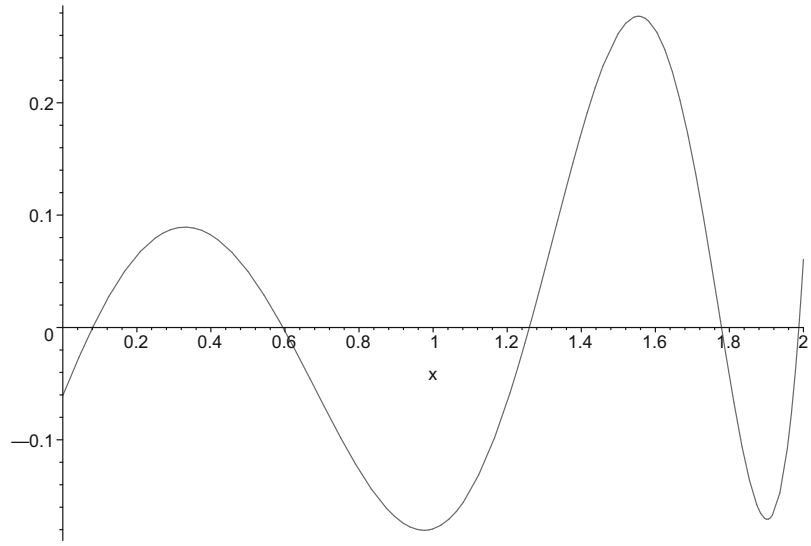


Figure 3.10 *Difference between* $P^{(1)}(x)$ *and* $\sin(\exp(x))$ *on* $[0, 2]$.

The difference $P^{(1)}(x) - \sin(\exp(x))$ is plotted in Figure 3.10.

We now compute the extremes of $P^{(1)}(x) - \sin(\exp(x))$ in $[0, 2]$, which gives the following new list of points: 0, 0.3305112886, 0.9756471625, 1.554268282, 1.902075854, 2. Solving the linear system associated to this list of points gives the polynomial

$$P^{(2)}(x) = 0.6800889007 + 2.144092090x$$
$$-1.631367834x^2 - 2.226220290x^3 + 1.276387351x^4.$$

The difference $P^{(2)}(x) - \sin(\exp(x))$ is plotted in Figure 3.11. One immediately sees that the extreme values of $|P^{(2)}(x) - \sin(\exp(x))|$ are very close together: $P^{(2)}$ "almost" satisfies the condition of

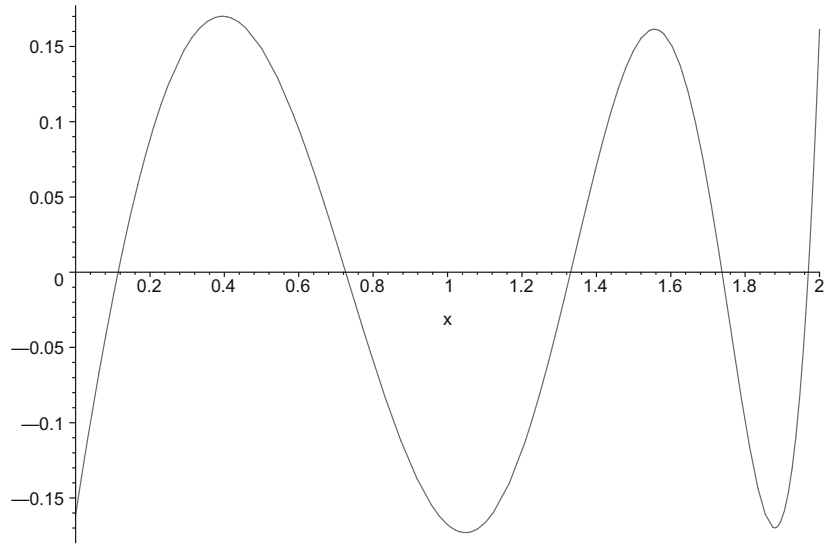


Figure 3.11 *Difference between* $P^{(2)}(x)$ *and* $\sin(\exp(x))$ *on* $[0, 2]$.

Theorem 11. This illustrates the fast convergence of Remez's algorithm: after two iterations, we already have a polynomial that is very close to the minimax polynomial.

Computing the extremes of $P^{(2)}(x) - \sin(\exp(x))$ in $[0, 2]$, gives the following new list of points: 0, 0.3949555564, 1.048154245, 1.556144609, 1.879537115, 2. From that list, we get the polynomial

$$P^{(3)}(x) = 0.6751785998 + 2.123809689x \\ -1.548829933x^2 - 2.293147068x^3 + 1.292365352x^4.$$

The next polynomial

$$P^{(4)}(x) = 0.6751752198 + 2.123585326x \\ -1.548341910x^2 - 2.293483579x^3 + 1.292440070x^4$$

is such that the ratio between the largest distance $|P^{(4)}(x) - \sin(\exp(x))|$ at one of the extremes and the smallest other distance is less than 1.000005: we can sensibly consider that we have found the minimax polynomial.

3.7 Minimizing the Maximum *Relative* Error

We have previously dealt with the best approximation, for the norm

$$||\cdot||_{\infty,[a,b]} = \max_{x\in[a,b]} |f(x) - P(x)|,$$

of a function f by a *polynomial* P of degree n, i.e., a linear combination of the monomials $1, x, x^2, x^3, \ldots, x^n$. Interestingly enough, most of what we have seen (Chebyshev's theorem, Remez's algorithm, etc.) can be relatively easily generalized to the approximation of a function by a linear combination of functions $\varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n$, provided that they satisfy the following condition:

Haar Condition *[52, 81] For any choice of* $n + 1$ *values* $x_0 < x_1 < x_2 < \cdots < x_n$ *in* $[a, b]$, *the determinant*

$$\begin{vmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \cdots & \varphi_n(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \varphi_1(x_n) & \cdots & \varphi_n(x_n) \end{vmatrix}$$

is nonzero. In the polynomial case ($\varphi_k(x) = x^k$), that determinant is the determinant of the linear system (3.5).

With choices such as $\varphi_k(x) = \cos(kx)$, Remez's algorithm has many important applications in signal processing. Here, we are mainly interested in choosing

$$\varphi_k(x) = \frac{x^k}{f(x)}, \tag{3.6}$$

since if these functions satisfy the Haar condition, then we can compute a best minimax approximation to the constant function 1 by a sum of the form

$$a_0 \cdot \frac{1}{f(x)} + a_1 \cdot \frac{x}{f(x)} + a_2 \cdot \frac{x^2}{f(x)} + \cdots + a_n \cdot \frac{x^n}{f(x)}.$$

By doing this, we find the values $a_0, a_1, \ldots, a_n$ that minimize

$$\max_{a \le x \le b} \left| 1 - \left(a_0 \cdot \frac{1}{f(x)} + a_1 \cdot \frac{x}{f(x)} + a_2 \cdot \frac{x^2}{f(x)} + \cdots + a_n \cdot \frac{x^n}{f(x)} \right) \right|,$$

which is equivalent to finding the polynomial $P^*(x) = a_0 + a_1 x + \cdots + a_n x^n$ that minimizes

$$\max_{a \le x \le b} \left| \frac{f(x) - P^*(x)}{f(x)} \right|,$$

i.e., we find the polynomial of degree less than or equal to n that minimizes the *maximum relative error* of approximation of f.

More generally, assume that one is interested in finding "the" polynomial P^* that minimizes

$$\max_{a \le x \le b} w(x) \cdot |f(x) - p(x)|,$$

where w is a continuous and positive weight function. The existence and unicity of P^* will be warranted if the functions $\varphi_k(x) = w(x) \cdot x^k$, for $0 \le k \le n$, satisfy the Haar condition, and the coefficients of P^* will be found using Remez's algorithm to obtain an approximation to $w(x) \cdot f(x)$ as a linear combination of the functions φ_k.

3.8 Rational Approximations

Table 3.4 gives the various errors obtained by approximating the square root on [0, 1] by polynomials. Even with degree-12 polynomials, the approximations are bad. A rough estimation can show that to approximate the square root on [0, 1] by a polynomial[5] with an absolute error smaller than 10^{-7}, one needs a polynomial of degree 54.

One could believe that this phenomenon is due to the infinite derivative of the square root function at 0. This is only partially true: a similar, yet less spectacular, phenomenon appears if we look for approximations on [1/4, 1]. The minimax degree-25 polynomial approximation to $\sqrt{x}$ on [1/4, 1] has an approximation error equal to 0.13×10^{-14}, whereas the minimax approximation of the same function by a rational function whose denominator and numerator have degrees less than or equal to 5 gives a better approximation error, namely, 0.28×10^{-15}. This shows that for some functions in some

[5]Of course, this is not the right way to implement the square root function: first, it is straightforward to reduce the domain to [1/4, 1], second, Newton–Raphson's iteration for $\sqrt{a}$:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right),$$

or (to avoid divisions) Newton–Raphson's iteration for $1/\sqrt{a}$:

$$x_{n+1} = \frac{x_n}{2}\left(3 - a x_n^2\right)$$

followed by a multiplication by a, or digit recurrence methods [179] are preferable.

Table 3.4 *Absolute errors obtained by approximating the square root on* [0, 1] *by a minimax polynomial.*

Degree	Error
4	0.034
5	0.028
6	0.023
7	0.020
8	0.017
9	0.016
10	0.014
11	0.013
12	0.012

domains, polynomial approximations may not be suitable. One has to try *rational approximations*.[6] Concerning rational approximations, there is a characterization theorem, similar to Theorem 11, that is also due to Chebyshev. Remind that $\mathcal{R}_{p,q}$ is the set of the rational functions with real coefficients whose numerator and denominator have degrees less than or equal to p and q, respectively.

Theorem 12 (Chebyshev) *An irreducible rational function $R^* = P/Q$ is the minimax rational approximation to f on $[a, b]$ among the rational functions belonging to $\mathcal{R}_{n,m}$ if and only if there exist at least*

$$k = 2 + \max\{m + degree(P), n + degree(Q)\}$$

values

$$a \leq x_0 < x_1 < x_2 < \cdots < x_{k-1} \leq b$$

such that

$$R^*(x_i) - f(x_i) = (-1)^i \left[R^*(x_0) - f(x_0)\right] = \pm||f - R^*||_\infty.$$

There exists a variant of Remez's algorithm for computing such approximations. See [225, 322, 388] for more details. Litvinov [322] notices that the problem of determining the coefficients is frequently "ill-posed," so the obtained coefficients may be quite different from the exact coefficients of the minimax approximation. And yet, it turns out that the computed fractions are high-quality approximants whose errors are close to the best possible. This is due to a phenomenon of "autocorrection," analyzed by Litvinov.

Another solution for getting rational approximations is to compute *Padé approximants* [26, 27], but such approximants have the same drawbacks as Taylor expansions: they are *local* (i.e., around one value) approximations only.[7] Algorithms that give "nearly best" approximations (even in regions of the complex plane) are given in [169]. There also exists a notion of *orthogonal rational functions* [71, 156]. See [35] for recent suggestions on rational approximation.

It seems quite difficult to predict if a given function will be much better approximated by rational functions than by polynomials. It makes sense to think that functions that have a behavior that is "highly nonpolynomial" (finite limits at $\pm\infty$, poles, infinite derivatives…) will be poorly approximated by polynomials.

[6] Another solution is to drastically reduce the size of the interval where the function is being approximated. This is studied in Chapter 6.

[7] And yet, they can have better global behavior than expected. See for instance reference [190].

Table 3.5 *Latencies of some floating-point instructions in double-precision/binary64 arithmetic for various processors, after [120, 144, 192, 421, 422].*

Processor	FP add	FP mult	FP div
Pentium III	3	5	32
Pentium IV	5	7	38
Intel Core i7 Nehalem	3	5	27
Intel Haswell	3	5	24
PowerPC 750	3	4	31
MIPS R10000	2–3	2–3	11–18
UltraSPARC III	4	4	24
Cyrix 5 × 86 and 6 × 86	4–9	4–9	24–34
Alpha21264	4	4	15
Athlon K6-III	3	3	20
AMD Bulldozer	5–6	5–6	42
AMD K10	4	4	31

For instance, the minimax degree-13 polynomial approximation of $\tan x$ in $[-\pi/4, +\pi/4]$ is

$$1.00000014609x + 0.333324808x^3 + 0.13347672x^5 + 0.0529139x^7 + 0.0257829x^9 + 0.0013562x^{11} + 0.010269x^{13}$$

with an absolute approximation error equal to 8×10^{-9}, whereas the minimax rational approximation with numerator of degree 3 and denominator of degree 4 of the same function is

$$\frac{0.9999999328x - 0.095875045x^3}{1 - 0.429209672x^2 + 0.009743234x^4}$$

with an absolute approximation error equal to 7×10^{-9}. In this case, to get the same accuracy, we need to perform 14 arithmetic operations if we use the polynomial approximation,[8] and 8 if we use the rational approximation.

Of course, the choice between polynomial or rational approximations highly depends on the ratio between the cost of multiplication and the cost of division. Table 3.5 gives typical figures for some processors. Those figures clearly show that for the moment, division is much slower than multiplication, so it is frequently preferable to use polynomial approximations.[9] This might change in the future: studies by Oberman and Flynn [364, 365] tend to show that fast division units could contribute to better performance in many areas.[10] As a consequence, future processors might offer faster divisions.

[8] Assuming that Horner's scheme is used, and that we first compute x^2.

[9] Unless some parallelism is available in the processor being used or the circuit being designed. For instance, as pointed out by Koren and Zinaty [278], if we can perform an addition and a multiplication simultaneously, then we can compute rational functions by performing in parallel an add operation for evaluating the numerator and a multiply operation for evaluating the denominator (and vice versa). If the degrees of the numerator and denominator are large enough, the delay due to the division may become negligible. However, we will see in Chapter 5 that the availability of parallelism also makes it possible to very significantly accelerate the evaluation of polynomials.

[10] The basic idea behind this is that, although division is less frequently called than multiplication, it is so slow (on most existing computers) that the time spent by some numerical programs in performing divisions is not at all negligible compared to the time spent in performing other arithmetic operations.

Table 3.6 *Errors obtained when evaluating* frac1 (x), frac2 (x), or frac3 (x) *in double-precision at* 500000 *regularly spaced values between* 0 *and* 1.

	frac1	frac2	frac3
Worst-case error	0.3110887e–14	0.1227446e–14	0.1486132e–14
Average error	0.3378607e–15	0.1847124e–15	0.2050626e–15

Another advantage of rational approximations is their flexibility: there are many ways of writing the same rational function. For instance, the expressions

$$\begin{aligned}
\text{frac1}\ (x) &= \frac{3 - 9x + 15x^2 - 12x^3 + 7x^4}{1 - x + x^2}, \\
\text{frac2}\ (x) &= 3 - 5x + 7x^2 - \frac{x}{1 - x + x^2}, \\
\text{frac3}\ (x) &= 3 + x \times \frac{-6 + 12x - 12x^2 + 7x^3}{1 - x + x^2},
\end{aligned}$$

represent the same function. One may try to use this property to find, among the various equivalent expressions, the one that minimizes the round-off error. This idea seems due to Cody [97]. It has been used by Hamada [213]. For instance, I evaluated the previous rational fraction in double-precision arithmetic (without using extended precision registers) using the three forms given previously, with the following parentheses (Pascal-like syntax):

```
function frac1(x: real):real;
begin
  frac1 := ((((7*x -12)*x+15)*x-9)*x+3) / ((x*x) - x + 1)
end;

function frac2(x:real):real;
begin
  frac2 := ((7*x -5 )*x+3) - (x / ((x*x) - x + 1))
end;

function frac3(x:real):real;
begin
  frac3 := 3 + (x * (((7 * x-12)*x+12)*x -6 ))
                  / ((x*x) - x + 1)
end;
```

The fraction was evaluated at 500000 regularly spaced values between 0 and 1, and compared with the exact result. The errors are given in Table 3.6. We immediately see that in [0, 1], expression frac2 is significantly better than frac1, and slightly better than frac3.

3.9 Accurately Computing Supremum Norms

Being able to compute the supremum norm

$$||g||_{\infty,[a,b]} = \max_{a \le x \le b} |g(x)|$$

of a continuous function is of uttermost importance. The polynomial of rational approximations we design will constitute the core of elementary function software that may sometimes be used in critical applications. If we approximate function f by a polynomial p, then

- underestimating the approximation error $||f - p||_{\infty,[a,b]}$ may lead to misbehavior: the software will be less accurate than what believed, so that some expected properties (such as correct rounding, or the absolute value of a sine being less than 1, or the preservation of monotonicity, etc.) may not be satisfied;
- overestimating the approximation error may lead to loss of performance: maybe the expected accuracy could have been achieved with a polynomial of lower degree, or a larger interval of approximation $[a, b]$ could have been used, resulting in a simpler and less often used range reduction.

The `infnorm` function of Maple is useful. It computes a tight approximation to the supremum norm of a function. It is part of the `numapprox` package.

For instance, the command lines

```
> Digits := 15:
>  infnorm(2^x - (0.999994405231621+0.693499150991505*x
+ 0.236778616969717* x^2 + 0.0661546610793498*x^3),x = 0..0.5);
```

ask for an approximation to $||2^x - p(x)||_{\infty,[0,1/2]}$, where

$$p(x) = 0.999994405231621 + 0.693499150991505 \cdot x$$
$$+0.236778616969717 \cdot x^2 + 0.0661546610793498 \cdot x^3.$$

the returned result is `0.000005595240536048 71`. It is an accurate *overestimate* of the actual value $0.000005595240535944426 5\cdots$.

On the other hand, the command lines

```
> Digits := 10:
> infnorm(sin(x)+exp(-x)-(1+x^2/2),x=0..1/4);
```

will return `0.005045257674`, which is a slight *underestimate* of the actual value $0.005045257674072202\cdots$.

For critical applications, one needs *certainty*: the best solution would be to obtain a very tight interval that is guaranteed to contain the supremum norm. This is what Sollya does. The Sollya package (available at http://sollya.gforge.inria.fr) was designed by Lauter et al. [85]. Among many interesting features, it offers a *certified* supremum norm of the difference between a polynomial and a function, that uses an algorithm designed by Chevillard, Harrison, Joldeş, and Lauter [83, 84, 86], and it computes nearly best polynomial approximations under constraints (using a method developed by Chevillard and Brisebarre [63])—this is an issue that we will investigate later on, in Chapter 4. By "certified" supremum norm, we mean that it provides a tight interval that is guaranteed to contain the supremum norm. The authors of [83] also wanted their method to allow for the possibility of generating a complete formal proof of their bounds without much effort, which is an important point if one wishes to certify a piece of software used in critical applications.

With Sollya, the first of the two examples we have examined is handled with the following command line:

```
> supnorm(0.999994405231621+0.693499150991505*x + 0.236778616969717* x^2
+ 0.0661546610793498*x^3,2^x,[0;0.5],absolute,2^(-80));
```

where `absolute` means that we are interested in absolute errors (one can also ask for relative errors), and the "`2^(-80)`" indicates the order of magnitude of the width of the interval that will enclose the supremum norm. Sollya returns

```
[5.59524053594442653392079259942630962509472021201 62e-6;
5.59524053594442653392079708306722559463502239871 83118e-6],
```

the supremum norm is guaranteed to lie in that interval. The algorithm used by Sollya is rather complex and the interested reader should have a look on [83]. Roughly speaking, an upper bound on $||f - p||_{\infty,[a,b]}$ is computed as follows:

- first, a very accurate, high-degree, polynomial approximation T to f is computed, for which we know an upper bound on $||T - f||_{\infty,[a,b]}$. Typically, T will be a Taylor expansion of f, for which we have an explicit bound on the approximation error, or what is called a "Taylor model" [259, 329]. T is chosen such that $||T - f||_{\infty,[a,b]}$ is much less than the expected (possibly approximately estimated using classical numerical methods) value of $||f - p||_{\infty,[a,b]}$;
- we know that $||f - p||_{\infty,[a,b]} \leq ||T - f||_{\infty,[a,b]} + ||T - p||_{\infty,[a,b]}$, so our problem is reduced to computing a bound on $||T - p||_{\infty,[a,b]}$: this is much easier than the initial problem since $T - p$ is a polynomial;
- now, if ϵ is the upper bound on $||T - p||_{\infty,[a,b]}$ we have hinted, actually proving that $||T - p||_{\infty,[a,b]}$ is less than ϵ reduces to proving the nonnegativity of the two polynomials $\epsilon - T + p$ and $\epsilon - p + T$ on $[a, b]$. There are several methods in the literature that allow one to show the nonnegativity of a polynomial, one of them (that is especially convenient for generating a simple formal proof) consists in expressing it as a sum of squares [221].[11]

An accurate and up-to-date presentation of rigorous methods for polynomial approximation is the Ph.D. dissertation of Joldeş [259].

3.10 Actual Computation of Approximations

It is no longer necessary to write specific software or to perform long paper and pencil calculations in order to compute polynomial or rational approximations of functions. Software such as Maple [78] readily computes minimax or Chebyshev approximations, and a tool such as Sollya [85] is entirely devoted to the approximation of functions (it computes certified polynomial approximations, supremum norms, plots functions, etc.). For instance, using Maple, the minimax polynomial approximants of degree 1 to 3 of the exponential function on [0, 1] and the corresponding errors are obtained as follows:

```
> with(numapprox);

[chebpade, chebyshev, confracform, hornerform, infnorm,
laurent, minimax, pade, remez, taylor]
------------------------------------------------------------
> Digits:=40;

                                  Digits := 40
------------------------------------------------------------
```

[11] A polynomial equal to a sum of squares of polynomials is nonnegative on all the real line, not only on $[a, b]$. To show nonnegativity on $[a, b]$ only, a change of variables may be necessary. Nonnegativity of the degree-n polynomial $P(x)$ for $x \in [a, b]$ is equivalent to nonnegativity of the polynomial $Q(y) = (1 + y^2)^n \cdot P((a + by^2)/(1 + y^2))$ for $y \in \mathbb{R}$. Hence, it is Q, not P that one should try to express as a sum of squares.

```
> for i from 1 to 3
> do
> minimax(exp(x),x=0..1,[i,0],1,'err');
> err;
> od;

      .894066583760558096609483029470243834207 5

           + 1.718281828459045235360287471352662497757 x

           .105933416239441903390516970529756165793


     1.008756022111995144557594671802439543032 + (

         .854742573413773377959371108234803856183 6

         + .846027210821281568285727019512979555510 4 x) x

              .008756022111995144557594671802439543032


    .999455208435667520950029003668877920315 7 + (

        1.016602326350653740263964831809759812173 + (

        .421703013129178039440604174270075170033 6

       + .279976488979213455655718465272827515552 x) x) x

          .000544791564332589764342588176415270745
```

The line

minimax(exp(x),x=0..1,[i,0],1,'err');

means that we are looking for a minimax approximation of the exponential function on [0, 1] by a rational function with a degree-i numerator and a degree-0 denominator (i.e., a degree-i polynomial !) with a weight function equal to 1, and that we want the variable `err` to be equal to the approximation error. From this example, one can see that the absolute error obtained when approximating the exponential function by a degree-3 minimax polynomial on [0, 1] is 5.4×10^{-4}.

Using Sollya,[12] the same minimax approximants of degree 1 to 3 of the exponential function on [0, 1] are obtained as follows:

```
> P1 = remez(exp(x),1,[0;1]);
> P2 = remez(exp(x),2,[0;1]);
> P3 = remez(exp(x),3,[0;1]);

> P1;
0.89406658399928255003969107406115435882562358497491
+ x * 1.71828182845904523536028747135266249775724709369998

> supnorm(P1,exp(x),[0;1],absolute,2^(-40));
[0.10593341651484907057524653684055238045402802526950 8;
0.10593341651494240564752434188556641864357905898811 2]
```

[12]The Sollya package is available at http://sollya.gforge.inria.fr.

```
> P2;
1.00875602211368932283041805215474175509109915735213
+ x * (0.85474257343306209250919106307541288356663921728 55
+ x * 0.84602721079860449719026030396776610400840956171021)

> supnorm(P2,exp(x),[0;1],absolute,2^(-40));
[8.75602211668725567987516672507908310763014014810 32e-3;
8.75602211669497037448253368830680557745374277306 88e-3]

> P3;
0.99945520842817029314254348598886281872860835239916
+ x * (1.01660232638589555870596955394194619558760370445
+ x * (0.42170301302379284328592780485228595873226086289258
+ x * 0.27997648904935683336839011255843034343738252635736 5))

> supnorm(P3,exp(x),[0;1],absolute,2^(-40));
[5.44791572016199348615949401425373466167911828961 22e-4;
5.44791572016679349756508701971903915689500382127 65e-4]
```

The line

```
> supnorm(P1,exp(x),[0;1],absolute,2^(-40));
```

means that we want Sollya to return an interval of size around 2^{-40} that contains the maximum value of $|P_1(x) - e^x|$ on $[0, 1]$. The term `absolute` means that we are interested in absolute error. If we had replaced that term by `relative`, Sollya would have computed the maximum value of $|(P_1(x) - e^x)/e^x|$.

Polynomial Approximations with Special Constraints

4

In the previous chapter, we have explained how Remez's algorithm can be used to compute minimax polynomial approximations to functions. Assume that we want to compute a polynomial approximation of degree n to some function f on interval $[a, b]$, in order to build a program that will be used later on for evaluating f. Also assume that we wish to use precision-p floating-point arithmetic in that program. There is no reason for the minimax polynomial approximation to have coefficients that fit exactly in precision p. In general, the coefficients of the "ideal," exact, minimax approximation to an elementary function on some interval are real numbers that cannot be exactly represented in any finite precision arithmetic. In practice, or course, Remez's algorithm is run using finite precision arithmetic, and if we want the obtained polynomial to be accurate enough, it is strongly advised to run Remez's algorithm in a precision q significantly larger than p. We therefore obtain a polynomial P with precision-q coefficients: how can we proceed to deduce an approximation with precision-p coefficients?

The first idea that springs to mind is to round each coefficient of P separately to the nearest precision-p floating-point number. This gives a new polynomial P_1. If P_1 is an approximation to f that is accurate enough for our purposes, there is no need to go further. However, it happens that $||f - P_1||_{\infty,[a,b]}$ is much larger than $||f - P||_{\infty,[a,b]}$. When this occurs, it means that we have lost a significant part of the quality of approximation when rounding the coefficients of P. Fortunately, in many cases, a better polynomial exists: *there is no reason for the best approximating polynomial with precision-p coefficients to be equal to the minimax polynomial with coefficients rounded to precision p*. We will see how such a better polynomial can be computed in Section 4.1, but in the meanwhile, let us consider an example.

Let P be the minimax degree-25 polynomial approximation to $\arctan(x)$ in $[0, 1]$, obtained with the Sollya command line

```
> P = remez(atan(x),25,[0;1]);
```

We have (using the `supnorm` command of Sollya)

$$||P - \arctan||_{\infty,[0,1]} \in [4.251 \cdot 10^{-19}, 4.252 \cdot 10^{-19}].$$

Let us separately round each of the coefficients of P to the nearest binary64 number. This gives a polynomial P_1. Sollya easily computes that polynomial

```
-8829299595920683 * 2^(-114) + x * (4503599627370499 * 2^(-52)
+ x * (-5159882142170469 * 2^(-95) + x * (-3002399751474879 * 2^(-53)
+ x * (-2703938931241909 * 2^(-82) + x * (1801439998904177 * 2^(-53)
```


J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_4

```
+ x * (-382606754743915 * 2^(-70) + x * (-40209430131455 * 2^(-48)
+ x * (-6987424401035279 * 2^(-67) + x * (8033486607699933 * 2^(-56)
+ x * (-1340893611715303 * 2^(-59) + x * (-5729819855457205 * 2^(-56)
+ x * (-6431143148353343 * 2^(-57) + x * (490042705331721 * 2^(-51)
+ x * (-805603107731421 * 2^(-51) + x * (5988052293807227 * 2^(-53)
+ x * (-5365331010775569 * 2^(-52) + x * (3546463639287455 * 2^(-51)
+ x * (-817810858300185 * 2^(-49) + x * (4134637789596061 * 2^(-52)
+ x * (-6664544602627907 * 2^(-54) + x * (1203328901139637 * 2^(-54)
+ x * (4942984695889011 * 2^(-58) + x * (-259019472710037 * 2^(-54)
+ x * (270784414594763 * 2^(-56)
+ x * (-7034512471776993 * 2^(-64))))))))))))))))))))))))).
```

We have

$$||P_1 - \arctan||_{\infty,[0,1]} \in [7.621 \cdot 10^{-17}, 7.622 \cdot 10^{-17}].$$

By rounding the coefficients to binary64 arithmetic, we have lost much accuracy: the approximation error is now around 179 times larger. Fortunately, P_1 is far from being the best degree-25 approximation to arctan, in [0, 1], with binary64 coefficients. Consider the following polynomial P_2 (obtained using a technique developed by Brisebarre and Chevillard [63], and implemented in the Sollya package[1])

```
-5597150764730203 * 2^(-113) + x * (4503599627370499 * 2^(-52)
+ x * (-5708274822853675 * 2^(-95) + x * (-6004799502933023 * 2^(-54)
+ x * (-89755795110261 * 2^(-77) + x * (900720003180065 * 2^(-52)
+ x * (-6377312872219227 * 2^(-74) + x * (-5146801287351833 * 2^(-55)
+ x * (-899685045909025 * 2^(-64) + x * (4017095024498131 * 2^(-55)
+ x * (-2742551287191647 * 2^(-60) + x * (-5713436721939779 * 2^(-56)
+ x * (-3272670461076033 * 2^(-56) + x * (3960850419627441 * 2^(-54)
+ x * (-6538895617370347 * 2^(-54) + x * (759706282506979 * 2^(-50)
+ x * (-5435319972600825 * 2^(-52) + x * (7182338603671099 * 2^(-52)
+ x * (-6635313536967241 * 2^(-52) + x * (8424322978787643 * 2^(-53)
+ x * (-429351662775197 * 2^(-50) + x * (2616431948988341 * 2^(-55)
+ x * (4303243131076383 * 2^(-58) + x * (-3973278851291135 * 2^(-58)
+ x * (8436312858769189 * 2^(-61)
+ x * (-6890517557888927 * 2^(-64))))))))))))))))))))))))).
```

We have

$$||P_2 - \arctan||_{\infty,[0,1]} \in [5.389 \cdot 10^{-19}, 5.390 \cdot 10^{-19}].$$

P_2 is more than 141 times better than P_1: the improvement is not negligible at all! There is a small loss of accuracy compared to P, but the polynomial P cannot be used "as is": its coefficients cannot be represented in binary64 arithmetic. What happens may be easier to see on graphs. Figure 4.1 is a plot of the difference $\arctan(x) - P(x)$. We see the equioscillating curve predicted by Chebyshev's Theorem (Theorem 11).

Figure 4.2 is a plot of $\arctan(x) - P_1(x)$. Its shape is totally different. By rounding the coefficients of P to the binary64 format, we have lost the equioscillating behavior.

Figure 4.3 is a plot of $\arctan(x) - P_2(x)$. Although it is not *exactly* equioscillating (otherwise it would be equal to the minimax polynomial, by Chebyshev's Theorem), it is extremely similar to the plot of Figure 4.1.

[1]There is no warranty that it is the *best* polynomial with binary64 coefficients. It is just a *very good* one.

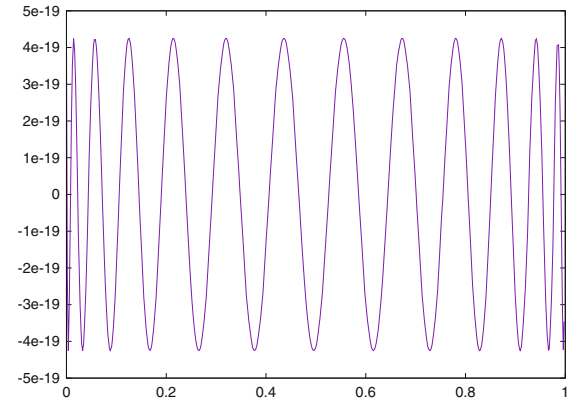


Figure 4.1 *Plot of the difference between* $\arctan(x)$ *and its degree-*25 *minimax approximation on* $[0, 1]$. *As predicted by Chebyshev's Theorem, the curve oscillates, and the extremum is attained* 26 *times (the leftmost and rightmost extrema are difficult to see on the plot, but they are here, at values* 0 *and* 1).

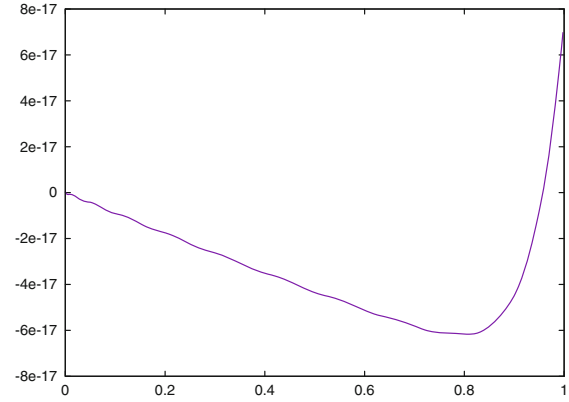


Figure 4.2 *Plot of the difference between* $\arctan(x)$ *and its degree-*25 *minimax approximation on* $[0, 1]$ *with coefficients rounded to the binary64 format. We have lost the "equioscillating" property, and the accuracy of approximation is much poorer.*

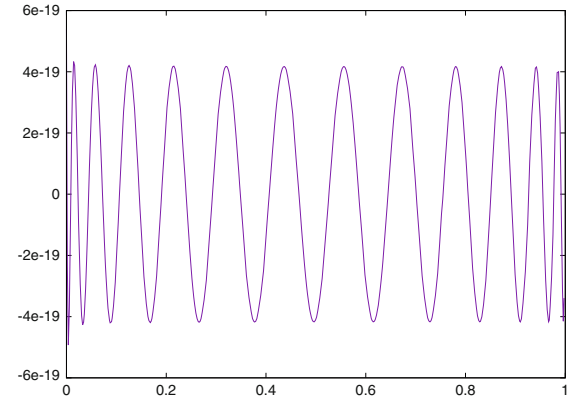


Figure 4.3 *Plot of the difference between* $\arctan(x)$ *and a degree-*25 *polynomial approximation with binary64 coefficients generated by Sollya. The accuracy of the approximation is almost as good as that of the minimax polynomial, and we have almost found again the "equioscillating" property.*

Importantly enough, one would probably not use Polynomial P_2 for actually implementing the arctan function. Indeed, another request one may have on the coefficients of a polynomial approximation is the following. If f is an even (resp. odd) function, one may wish its approximating polynomial to be even (resp. odd) too. This preserves symmetry and makes the number of multiplications used for evaluation smaller). That request is not so difficult to satisfy. As a matter of fact, the minimax approximation to an even (resp. odd) function on an interval *that is symmetrical around zero* is an even (resp. odd) polynomial. It is easy to convince oneself of that property: if $P(x)$ is the best polynomial approximation

to $\sin(x)$ on $[-\pi/2, +\pi/2]$, and if $||P - \sin||_{\infty,[-\pi/2,\pi/2]} = \epsilon$, then $Q(x) = -P(-x)$ too is a polynomial approximation to $\sin(x)$ on $[-\pi/2, +\pi/2]$ with maximum error ϵ. Since the minimax approximation is unique, we necessarily have $Q(x) = P(x)$. Notice that the minimax polynomial approximation to an even (resp. odd) function numerically computed in practice will have tiny yet nonzero odd-order (resp. even-order) coefficients. These nonzero coefficients are just numerical noise, and can be discarded. When computing the minimax approximation P to $sin(x)$ on $[-\pi/2, \pi/2]$ using the `remez` function of Sollya, we get

```
-1.4198552932567808e-16 + x * (0.99969677313904359
+ x * (1.3389005866132525e-15 + x * (-0.16567307932054534
 + x * (-1.1305009912441752e-15 + x * 7.5143771782993267e-3))))
```

and the approximation error lies in the interval

$$[6.7706 \cdot 10^{-5}, 6.7707 \cdot 10^{-5}].$$

We effectively have very tiny even-order coefficients. They are just due to roundoff errors. If we discard them, we obtain the following polynomial P_1:

```
P1 = x * (0.99969677313904359 + x^2*(-0.16567307932054534
+ x^2* 7.5143771782993267e-3));
```

and using the `supnorm` function of Sollya, we find that

$$||P_1 - \sin||_{\infty,[-\pi/2,+\pi/2]} \in [6.7706 \cdot 10^{-5}, 6.7707 \cdot 10^{-5}],$$

i.e., it is the same approximation error.

However, even if the final goal is to use the approximation for positive values of x only, one should not start from an approximation to $\sin(x)$ in $[\mathbf{0}, \pi/2]$ (i.e., we really need a interval that is symmetrical around 0). When computing the minimax approximation P to $\sin(x)$ on $[0, \pi/2]$ using the `remez` function of Sollya, we obtain the following polynomial P_2:

```
7.0685185941109779e-6 + x * (0.999689864433651
+ x * (2.1937161796562626e-3 + x * (-0.17223886510877676
+ x * (6.0973836901399977e-3 + x * 5.7217240503332698e-3))))
```

The error $||P_2 - \sin||_{\infty,[0,+\pi/2]}$ lies in $[7.0685 \cdot 10^{-6}, 7.0686 \cdot 10^{-6}]$ (it is smaller than $||P - \sin||_{\infty,[0,+\pi/2]}$, which is not surprising since the interval of approximation is smaller). However, if we discard in P_2 the even-order coefficients to get a new polynomial P_3, we have

$$||P_3 - \sin||_{\infty,[-\pi/2,+\pi/2]} \in [4.2534 \cdot 10^{-2}, 4.2535 \cdot 10^{-2}],$$

i.e., the approximation is now very poor.

A last request one may have is to impose the value of a very few low-order coefficients (typically, the value would be the value of the corresponding coefficients in the Taylor expansion of the function), in order to impose the behavior of the approximation near zero. Such approximations have been studied by Dunham [159, 161, 162, 164, 165, 166].

Combining all these requests leads for instance to trying to find approximations to the sine function of the form $x + x^3 p(x^2)$ where the coefficients of p are floating-point numbers.

4.1 Polynomials with Exactly Representable Coefficients

Let us now show how we can compute polynomial approximations with coefficients that are exactly representable in a given precision. We will briefly present three methods that have been suggested to deal with this problem. Assume we want to approximate function f on interval $[a, b]$.

4.1.1 An Iterative Method

As explained above, it is important to get polynomial approximations whose coefficients are floating-point numbers in the desired format,[2] and in many cases the polynomial obtained by just rounding the coefficients of the minimax polynomial is not a fully satisfying approximation. The following method was probably invented by Kahan and his students (it was not published in the open literature, but it is part of the computer arithmetic folklore).

Compute the minimax approximation to f on $[a, b]$ with extra precision, round the coefficient of order 0 to the nearest number in the desired format, say a_0, then recompute an approximation $P^{(1)}$, where you impose the order 0 coefficient to be a_0.

Define $\hat{P}^{(1)}$ as the polynomial obtained by rounding all the coefficients of $P^{(1)}$. If the approximation error

$$||\hat{P}^{(1)} - f||_{\infty,[a,b]}$$

is not significantly larger than $||P^{(1)} - f||_{\infty,[a,b]}$ then $\hat{P}^{(1)}$ is the desired polynomial. Otherwise, define a_1 as the coefficient of order 1 of $\hat{P}^{(1)}$ (equal to the coefficient of order 1 of $P^{(1)}$ rounded), and compute an approximation $P^{(2)}$, where you impose the first two coefficients to be a_0 and a_1. Continue until there is no significant difference between $||\hat{P}^{(i)} - f||_{\infty,[a,b]}$ and $||P^{(i)} - f||_{\infty,[a,b]}$ (i.e., until rounding the coefficients has no significant effect).

Example 3 *(Computation of 2^x) Assume that we wish to approximate 2^x on $[0, 1/32]$ by a polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$ of degree 3, and that we plan to use the approximation in IEEE-754 binary32/single-precision arithmetic. Using the* `minimax` *function of Maple, we first compute, in a significantly wider precision arithmetic the minimax approximation*

```
.99999999992755851125495676128 5+(.693147254659769878047982577583
+(.240214666378173867533469143950
+0.561090023893935052398844196228e-1*x)*x)*x.
```

After this, we impose the coefficient of degree 0 *to be*

$$a_0 = \mathrm{RN}(0.99999999992755851125495676128 5) = 1.$$

If ϵ is the approximation error of the new approximation we wish to compute, and if we define $p(x)$ to be the degree-2 polynomial $a_1 + a_2x + a_3x^2$, we want

$$\left|\left(2^x - a_0\right) - xp(x)\right| \leq \epsilon.$$

[2] Or the sum of two machine numbers, at least for the leading coefficients, when very high accuracy is at stake.

This is equivalent to

$$\left|\frac{2^x - a_0}{x} - p(x)\right| \cdot x \le \epsilon.$$

Therefore it suffices to compute a degree-2 minimax approximation to

$$\frac{2^x - a_0}{x}$$

with a weight function x. In Maple, this is done with the command line

```
minimax((2^x-1)/x,x=0..1/32,[2,0],x,'err');
```

and this gives

```
.693147234930309852689080196056+(.240215961912910986166070981685
+0.560849766217597431698702570574e-1*x)*x
```

We continue by choosing $a_1 = \mathrm{RN}(0.69314723493030985268908019605 6) = 11629081/16777216$*, and by looking for a degree-*1 *minimax approximation to*

$$\frac{2^x - a_0 - a_1 x}{x^2}$$

with a weight function x^2. This gives

```
.240215057046252382183162007004+0.561094355336852235183806436094e-1*x.
```

We now choose
$a_2 = \mathrm{RN}(0.240215057046252382183162007004) = 16120527/67108864.$
The next step gives $a_3 = 7533447/134217728$*, and we check that*

$$\left|\left|2^x - \left(a_0 + a_1 x + a_2 x^2 + a_3 x^3\right)\right|\right|_{\infty,[0,1/32]} \approx 1.454 \cdot 10^{-10}.$$

We can easily check that if we had just rounded each of the coefficients of the initial polynomial to the nearest binary32 floating-point number, the approximation error would have been around $3.988 \cdot 10^{-10}$.

4.1.2 An Exact Method (For Small Degrees)

In general, the method we have just dealt with does not give *best* approximations, it only gives *good* ones. Let us now describe how to get such best approximations. The following method was suggested by Brisebarre, Muller, and Tisserand [69]. It works when the degree of the polynomial is not too large. Assume we wish to find the best polynomial approximation $p^*(x) = p_0^* + p_1^* x + \cdots + p_n^* x^n$ to $f(x)$ in $[a, b]$, with the constraint that p_i^* must be a multiple of 2^{-m_i} (that is, its binary representation has at most m_i fractional bits). Define p as the usual minimax approximation to f in $[a, b]$ without that constraint, and $\hat{p}$ as the polynomial obtained by rounding the degree-i coefficient of p to the nearest multiple of 2^{-m_i}, for all i. Define

$$\epsilon = \max_{x\in[a,b]}|f(x) - p(x)|$$
$$\hat{\epsilon} = \max_{x\in[a,b]}|f(x) - \hat{p}(x)|.$$

We obviously have

$$\epsilon \leq \max_{x\in[a,b]}|f(x) - p^*(x)| \leq \hat{\epsilon}.$$

The technique suggested in [69] consists in first finding a *polytope* (i.e., a bounded polyhedron) where p^* necessarily lies,[3] and then to scan all possible candidate polynomials (a candidate polynomial is a degree-n polynomial whose degree i coefficient is a multiple of 2^{-m_i} for all i, and that lies inside the polytope. These constraints imply that the number of candidate polynomials is finite) using recent scanning algorithms [7, 105, 451], and computing the distance to f for each of these polynomials. If we look for a polynomial p^* such that

$$\max_{x\in[a,b]}|f(x) - p^*(x)| \leq K,$$

then one can build a polytope by choosing at least $n + 1$ points

$$a \leq x_0 < x_1 < x_2 < \cdots < x_m \leq b, m \geq n$$

and imposing the linear[4] constraints[5]

$$f(x_j) - K \leq p_0^* + p_1^* x_j + p_2^* x_j^2 + \cdots + p_n^* x_j^n \leq f(x_j) + K, \quad \forall j. \tag{4.1}$$

If $K < \epsilon$, then the polytope defined by (4.1) contains no candidate polynomial. If $K \geq \hat{\epsilon}$, it contains at least one candidate polynomial (that is, $\hat{p}$), and in practice it may contain too many candidate polynomials, which would make the scanning of the polytope very long. In practice, one has to start the algorithm with K significantly less than $\hat{\epsilon}$. See [69] for more details.

Roughly speaking, the number of candidate polynomials to be examined will increase with the degree exponentially. This makes the method unpractical for polynomial approximations of large degrees. The next method will not, in general, return the *best* approximation with exactly representable coefficients, but it will always return a *good* one.

4.1.3 A Method Based on Lattice-Reduction

The following method is due to Brisebarre and Chevillard [63]. It was implemented in the Sollya software. Since its presentation is rather technical, the reader who is just interested in using the method can skip this section and directly go to Section 4.2.

[3] The polytope is located in a space of dimension $n+1$. A degree-n polynomial is a point in that space, whose coordinates are its coefficients.

[4] Linear in the coefficients p_i^*.

[5] In practice, we use slightly different constraints: we modify (4.1) so that we can work with rational numbers only.

Euclidean lattices

Let $b_1, b_2, \ldots, b_d$ be linearly independent elements of $\mathbb{R}^d$. The *Euclidean lattice* L (called *Lattice* for short) generated by $(b_1, b_2, \ldots, b_d)$ is the set of the linear combinations of the vectors b_i with *integer* coefficients

$$L = \{\lambda_1 b_1 + \lambda_2 b_2 + \cdots + \lambda_d b_d \mid \lambda_1, \lambda_2, \ldots, \lambda_d \in \mathbb{Z}\}.$$

We will say that $(b_1, b_2, \ldots, b_d)$ is a basis of L, and d is the dimension of the lattice.

Euclidean lattices have many interesting applications in number theory and cryptography. They also have been used for finding hardest-to-round cases for the most common elementary functions (see Chapter 12 and [432, 433]). A lattice of dimension larger than or equal to 2 has infinitely many bases. However, they are not equally interesting from an algorithmic point of view. For many applications, we need bases made up with vectors that are as small as possible. An algorithm called LLL (after the names of its inventors, Lenstra, Lenstra, and Lovász [313]) computes a basis with reasonably short vectors (called a reduced basis) in polynomial time.

Assume $||\cdot||$ is a norm on $\mathbb{R}^n$. Many problems linked with lattices reduce to the following problem, called *Closest vector problem* (CVP):

> Let L be a lattice of dimension d (given by one of its bases), and let x be an element of $\mathbb{R}^d$. Find $y \in L$ such that $||x - y|| = \min_{z \in L} ||x - z||$.

In practice, that problem is much too hard to solve (it is NP-hard). Instead of trying to solve CVP, we try to solve the following associated approximation problem:

> Find $y \in L$ such that $||x - y|| \leq \gamma \cdot \min_{z \in L} ||x - z||$, where $\gamma > 1$ is fixed.

Babai [14] introduced an algorithm for solving that approximation problem, with $\gamma = 2^{d/2}$. Babai's algorithm uses a reduced basis of the lattice (hence, we must first run the LLL algorithm).

Application to Our Problem

Our problem of finding degree-n polynomial approximations with exactly representable coefficients can be rewritten as follows: given integers $m_1, m_2, \ldots, m_n$, find a polynomial

$$p^* = \frac{a_0}{2^{m_0}} + \frac{a_1}{2^{m_1}} + \cdots + \frac{a_n}{2^{m_n}},$$

where the a_i are integers that minimize (or at least make small) $||f - p^*||_{\infty,[a,b]}$ (see [63] for an explanation of why we can fix the value of the m_i's). We can discretize that problem: if $x_1, x_2, \ldots, x_\ell$ are adequately chosen points of $[a, b]$, we want each of the values $p^*(x_i)$ to be as close as possible to $f(x_i)$, that is we want the dimension-ℓ vector

$$a_0 \cdot \begin{pmatrix} \frac{1}{2^{m_0}} \\ \frac{1}{2^{m_0}} \\ \vdots \\ \frac{1}{2^{m_0}} \end{pmatrix} + a_1 \cdot \begin{pmatrix} \frac{x_1}{2^{m_1}} \\ \frac{x_2}{2^{m_1}} \\ \vdots \\ \frac{x_\ell}{2^{m_1}} \end{pmatrix} + \cdots + a_n \cdot \begin{pmatrix} \frac{x_1^n}{2^{m_n}} \\ \frac{x_2^n}{2^{m_n}} \\ \vdots \\ \frac{x_\ell^n}{2^{m_n}} \end{pmatrix}$$

to be as close as possible to the vector

$$\begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}.$$

Clearly, this is an instance of CVP. Brisebarre and Chevillard suggest to choose as points x_i finite precision values as close as possible to the (real) values where f and its minimax polynomial approximation are equal (Chevyshev's theorem implies that there are at least $n+1$ such points). They use Babai's algorithm to solve an approximation to the CVP problem. Their method, implemented in Sollya, works well and allows to tackle approximations of large degrees.

4.2 Getting Nearly Best Approximations Using Sollya

As a first example, assume that we wish to compute a polynomial approximation to function 2^x in the interval $[0, 1/32]$. Under the Sollya environment, we can type

```
> Premez = remez(1,3,[0;1/32],2^(-x));
```

which means that we want to find the polynomial p^* of degree less than or equal to 3 that minimizes

$$\max_{x\in[0,1/32]} \left|p^*(x)\cdot 2^{-x} - 1\right| = \max_{x\in[0,1/32]} \left|\frac{p^*(x) - 2^x}{2^x}\right|,$$

using Remez's algorithm. We can display the obtained polynomial by typing

```
> Premez;
```

which gives

```
0.99999999992833969012490146234825301922563977887262
+ x * (0.69314725420872850397378954914175310756621257309927
+ x * (0.24021470491338404661352112040401957279342133236739
+ x * 5.61081797109160130345894929451981381992035061491 26e-2)).
```

The error of that approximation is obtained as follows:

```
> supnorm(Premez,2^x,[0;1/32],relative,2^(-40));
```

where "`relative`" indicates that we want to compute a relative error, and "2^{-40}" gives to Sollya an order of magnitude on the tightness of the returned interval that contains the maximum of the approximation error. The obtained result is

```
[7.16603098983067960910058050611060472104328969433 77e-11;
     7.16603098983699340546800988381757733704517932083 64e-11].
```

The obtained result is a *certified* enclosure of the approximation error. We can compare the obtained result with the output of the following sequence of Maple commands:

```
Digits := 70:
P := x -> 0.99999999992833969012490146234825301922563977887262
+ x * (0.69314725420872850397378954914175310756621257309927
```

```
+ x * (0.24021470491338404661352112040401957279342133236739
+ x * 5.61081797109160130345894929451981381992035061491 26e-2)):
infnorm((P(x)-2^x)/2^x,x=0..1/32);
```

which returns

$$7.16603098983067538181452733796390234568596287381805621489873670879509 9 \cdot 10^{-11}.$$

Here, the `infnorm` function of Maple gives a result that is very close to the maximum approximation error, but that very slightly *underestimates* it. This requires caution if one wishes to build elementary function software with guaranteed error bounds.

Now, let us ask Sollya to build an approximation with constraints. For example, by typing

```
> P1 := fpminimax(2^x,3,[|1,24...|],[0;1/32],relative);
```

we ask for a degree-3 polynomial approximation to 2^x in the interval $[0, 1/32]$ that minimizes the maximum relative error, with a first (i.e., degree-0) coefficient that fits in a 1-bit-significand floating-point number (which is in practice a way of requiring that coefficient to be equal to 1), and the other coefficients that fit in 24-bit-significand floating-point numbers (i.e., they are required to be single-precision/binary32 floating-point numbers). Let us display the obtained polynomial

```
> P1;
1 + x * (0.693147242069244384765625
+ x * (0.240215316414833068847656 25
+ x * 5.609864741563796997070312 5e-2))
```

We can display it in a somehow more readable form by typing

```
> display=powers!;
> P1;
1 + x * (11629081 * 2^(-24)
+ x * (16120577 * 2^(-26)
+ x * (7529433 * 2^(-27))))
```

The approximation error is obtained as previously:

```
> supnorm(P1,2^x,[0;1/32],relative,2^(-40));

[1.024905796492459173179126919958958555475700413772 7e-10;
      1.024905796493362189996111729137367604970240064232 3e-10]
```

We see that even after having required very strong constraints on the polynomial (the first coefficient is 1, and the other ones are single-precision numbers), we have not lost that much on the approximation error.

Now, let us try to obtain a much more accurate approximation to 2^x. To be able to return *correctly rounded* results (see Chapter 12, and more precisely Table 12.6), let us assume that we wish to approximate function 2^x in the interval $[0, 1/64]$ with relative accuracy better than $2^{-114} \approx 4.8 \times 10^{-35}$. We will use a polynomial of degree 10 (a call to `remez` with degree 9 shows that we need at least degree 10). We still impose the coefficient of degree 0 to be equal to 1 (this is again obtained by requiring the significant of that coefficient to fit in one bit). Let us start by imposing all other coefficients to be "double-double" numbers, or "double-word numbers" in the double-precision/binary64 format (that is, we require them to be representable as the unevaluated sum of two double-precision/binary 64 floating-point numbers—see Section 2.2.2). In Sollya, this is done by typing

```
> P2 = fpminimax(2^x,10,[|1,DD...|],[0;1/64],relative);
```

(the "DD" stands for "double-double") followed—to know the relative approximation error—by

```
> supnorm(P2,2^x,[0;1/64],relative,2^(-40));
```

we get the following result:

```
[3.07935289575623741430997024839975938177274604398 31e-36;
     3.07935289575895054898031281358204367343006806025 59e-36]
```

which shows that the obtained polynomial suffices for our purposes (provided that when we add the *evaluation* error to that *approximation* error we do not exceed 2^{-114}: we will deal with evaluation errors in Chapter 5). However, since the interval under consideration is $[0, 1/64]$, the contribution of the coefficients of high degree to the final result is small: perhaps it is useless to represent these coefficients with very high precision. Hence, let us try to require only the coefficients of degrees 1, 2, and 3 to be double-double numbers, and the coefficients of higher degrees to be double-precision numbers. This is done by typing

```
> P3 = fpminimax(2^x,10,[|1,DD,DD,DD,D...|],[0;1/64],relative);
```

the error estimated by `supnorm` is

```
[4.884954769406162093652511345764775013204486435216 25e-32;
      4.884954769410466095368649918437187815231560671349 8e-32]
```

which shows that the obtained polynomial is not accurate enough. If we allow the coefficient of degree 4 too to be a double-double number, that is, if we type

```
> P4 = fpminimax(2^x,10,[|1,DD,DD,DD,DD,D...|],[0;1/64],
           relative);
```

then the error estimated by `supnorm` is

```
[4.053612018666920511187148838345936645549024807571 1e-36;
    4.053612018670492039258231322189513629364341829848 2e-36]
```

we now have a polynomial that suits our purposes.

If a function is even (resp. odd), we can ask Sollya to compute polynomial approximations with coefficients of even order (resp. odd order) only. For instance, the polynomial

$$1 - 2251799813622611 \cdot 2^{-52} \cdot x^2 + 1501189276987675 \cdot 2^{-55} \cdot x^4$$

that was used for building the example of Section 2.2.4 is an approximation to $\cos(x)$ in the interval $[-0.0123, +0.0123]$. It was generated by typing the following line in Sollya:

```
> P := fpminimax(cos(x),[|0,2,4|],[|1,D,D|],[-0.0123;0.0123],
           relative);
```

where `[|0,2,4|]` indicates that we want a polynomial made up with monomials of degrees 0, 2, and 4 only (i.e., a degree-4 polynomial whose coefficients of degrees 1 and 3 are zero). Sollya tells us that the maximum relative approximation error of this polynomial is in the interval

$$[1.899908785048 \cdot 10^{-16}, 1.899908785051 \cdot 10^{-16}].$$

Now, let us try to do something similar with an odd function: let us try to approximate $\sin(x)$, for $x \in [0, \pi/8]$, by an odd polynomial of degree 5. If we type the command

```
> Q := fpminimax(sin(x),[|1,3,5|],[|D...|],[0;pi/8],relative);
```

then Sollya fails to find an approximation. Fortunately, we have two possible turn-arounds. The first one is the following: if Q is an odd polynomial, then it can be written $P \cdot x$, where P is an even polynomial, and

$$\frac{\sin(x) - Q(x)}{\sin(x)} = \epsilon$$

is equivalent to

$$\frac{\frac{\sin(x)}{x} - P(x)}{\frac{\sin(x)}{x}} = \epsilon,$$

which means that approximating $\sin(x)$ by $Q(x)$ with some maximum relative error is equivalent to approximating $\sin(x)/x$ by $P(x)$ with the same maximum relative error. Now, if we type

```
> P := fpminimax(sin(x)/x,[|0,2,4|],[|1,D...|],[0;pi/8],relative);
```

we obtain

$$P(x) = 1 - 1501181244204129 \cdot 2^{-53} \cdot x^2 + 298697310996351 \cdot 2^{-55} \cdot x^4,$$

which means that

$$Q(x) = x - 1501181244204129 \cdot 2^{-53} \cdot x^3 + 298697310996351 \cdot 2^{-55} \cdot x^5.$$

We can now obtain the enclosure of the maximum relative approximation error by typing

```
> supnorm(Q,sin(x),[0;pi/8],relative,2^(-40));
```

and conclude that $Q(x)$ approximates $\sin(x)$ in $[0, \pi/8]$ with a maximum relative error in

$$[2.903562688025 \cdot 10^{-8}, 2.903562688028 \cdot 10^{-8}].$$

The second turn-around—much simpler to implement—is the following. The initial problem is due to the fact that Sollya probably had difficulties to notice that the relative error $(\sin(x) - p(x))/\sin(x)$ can have a finite limit at zero. Assume the variable x of the polynomial we are building is a binary64 floating-point number. It will be either zero or a number of magnitude larger than 2^{-1074}. Hence, we can try the Sollya command line

```
> Q := fpminimax(sin(x),[|1,3,5|],[|1,D...|],[2^(-1074);pi/8],
            relative);
```

and we will get exactly the same polynomial as with the other turn-around. These turn-arounds look suspicious? This is not a problem: Sollya provides a certified supremum norm, therefore all strange tricks can be used to find the approximating polynomial: what matters is that ultimately we can safely check, with the certified supremum norm, that the approximation error is within the desired range. Sollya has been used by several authors to build convenient approximations to elementary [452, 70] or special [299] functions, or for more general applications such as spacecraft control [142].

4.3 Miscellaneous

Polynomial and rational approximations of functions have been widely studied [90, 93, 94, 140, 160, 169, 188, 189, 196, 225, 278, 388, 394]. Good references on approximation are the books by Cheney [80], Rice [396], Rivlin [397], Laurent [298], Phillips [385], Powell [388], and Trefethen [458]. Some of the ideas presented in this chapter can be extended to complex elementary functions. Braune [54] and Krämer [284] suggested algorithms that evaluate standard functions and inverse standard functions for real and complex point and interval arguments with dynamic accuracy. Midy and Yakovlev [345] and Hull et al. [242] suggested algorithms for computing elementary functions of a complex variable. The CELEFUNT package, designed by W.J. Cody [101], is a collection of test programs for the complex floating-point elementary functions: now it is outdated but many ideas are still of interest. Minimax approximation by polynomials on the unit circle is considered in [23, 460], with applications to digital filtering. When evaluating a given polynomial may lead to underflow or overflow, there exist scaling procedures [214] that prevent overflow and reduce the probability of occurrence of underflow.

Polynomial Evaluation

5

In Chapters 3 and 4, we have studied how a function can be approximated by a polynomial or a rational function. When actually implementing the approximation, one has to select the way to evaluate a polynomial in order to minimize the error and/or to maximize the speed. The adequate choices may differ depending on whether we wish to design a "general" function library, that will be used on different processor architectures, or a function library aimed to best work on a particular architecture or family of architectures for which basic properties (amount of available parallelism, pipeline depths, availability of an FMA instruction…) are known. An important current trend in computer arithmetic research consists in trying to automate as much as possible the choice of the evaluation algorithm [351].

5.1 Sequential Evaluation of Polynomials

Let us temporarily assume that no parallelism is available. On modern architectures this is an unrealistic assumption: even a single pipelined arithmetic operator has an inherent parallelism, since it can start performing an operation before the previous one has terminated. We will deal with parallel evaluation of polynomials later on, in Section 5.2. Before going further, here is some elementary advice *never* evaluate a polynomial

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

using the sequence of operations

```
a_4*x*x*x*x + a_3*x*x*x + a_2*x*x + a_1*x + a_0;
```

there are evaluation methods that are faster. The simplest and most popular is Horner's scheme, presented below.

5.1.1 Horner's Scheme

Horner's scheme consists in evaluating a polynomial

$$a_nx^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0, \tag{5.1}$$

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_5

by the following algorithm:

Algorithm 13 Horner's scheme.

$y \leftarrow a_n \cdot x + a_{n-1}$
for $i = n - 2$ downto 0 **do**
  $y \leftarrow y \cdot x + a_i$
end for
return y

which corresponds to the following way of parenthesizing (5.1):

$$(((\cdots(a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0.$$

Although commonly attributed to Horner [236], this method was used previously by Newton [275], and seems to have been known 2000 years before Horner by Chinese mathematicians of the Han dynasty [473]. To evaluate a polynomial of degree n, Horner's scheme requires n multiplications and n additions. This was shown to be optimal if the coefficients a_i are not known in advance (which of course is not really the case here): more precisely, Ostrowski [372] showed that we need at least n additions,[1] and Pan [374] showed that we need at least n multiplications. See also [392]. If the a_i's are known in advance, some improvement is possible (a method due to Knuth is presented in Section 5.1.2), but the improvement is somehow limited: there are degree-n polynomials for which we need at least $n/2$ multiplications[2]. Hence, if no parallelism is available, Horner's scheme is a good solution. Notice that most recent processors have a "fused multiply–add" (FMA) instruction (see Section 2.1.5); that is, an expression of the form $a \times x \pm b$ can be evaluated just with one instruction, and there is only *one* rounding error at the end. That instruction allows one to implement Horner's scheme very efficiently.

5.1.2 Preprocessing of the Coefficients

If the degree of the polynomial is large,[3] one can use a method called "adaptation of coefficients," that was analyzed by Knuth [275]. This method consists of computing once and for all some "transformation" of the polynomial that will be used later on for evaluating it using fewer multiplications than with Horner's scheme. It is based on the following theorem.

Theorem 13 (Knuth) *Let $u(x)$ be a degree-n polynomial*

$$u(x) = u_n x^n + u_{n-1} x^{n-1} + \cdots + u_1 x + u_0.$$

[1] Even in the very simple case when $x = 1$ and we just have to compute $a_0 + \cdots + a_n$. Of course we immediately see that there is a simplification if the a_i's are known in advance!

[2] Even if specific polynomials, such as x^n, can be evaluated with much fewer multiplications.

[3] Which sometimes occurs, see for instance [115], page 267.

Let $m = \lceil n/2 \rceil - 1$. There exist parameters c, $\alpha_0, \alpha_1, \ldots, \alpha_m$ and $\beta_0, \beta_1, \ldots, \beta_m$ such that $u(x)$ can be evaluated using at most $\lfloor n/2 \rfloor + 2$ multiplications and n additions by performing the following calculations:

$$\begin{aligned}
y &= x + c \\
w &= y^2 \\
z &= (u_n y + \alpha_0)y + \beta_0 \textit{ if } n \textit{ is even} \\
z &= u_n y + \beta_0 \textit{ if } n \textit{ is odd} \\
u(x) &= (\ldots((z(w - \alpha_1) + \beta_1)(w - \alpha_2) + \beta_2)\ldots)(w - \alpha_m) + \beta_m.
\end{aligned}$$

For instance, if $n = 7$ and if we choose $c = 1$, the values $\alpha_0, \alpha_1, \ldots, \alpha_m$ and $\beta_0, \beta_1, \ldots, \beta_m$ can be obtained by solving the following system of equations:

$$\left\{\begin{aligned}
u_7 &= 8u_8 + \alpha_0 \\
u_6 &= 25u_8 + 7\alpha_0 + \beta_0 + u_8(1-\alpha_1) + u_8(1-\alpha_2) + u_8(1-\alpha_3) \\
u_5 &= 38u_8 + 18\alpha_0 + 6\beta_0 + (2u_8 + \alpha_0)(1-\alpha_1) + 4u_8(1-\alpha_1) \\
&\quad +(4u_8 + \alpha_0)(1-\alpha_2) + 2u_8(1-\alpha_2) + (6u_8 + \alpha_0)(1-\alpha_3) \\
u_4 &= (u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + \beta_1 + 28u_8 + 20\alpha_0 + 12\beta_0 \\
&\quad +4(2u_8 + \alpha_0)(1-\alpha_1) + (5u_8 + 3\alpha_0 + \beta_0 + u_8(1-\alpha_1))(1-\alpha_2) \\
&\quad +4u_8(1-\alpha_1) + 2(4u_8 + \alpha_0)(1-\alpha_2) + \\
&\quad (13u_8 + 5\alpha_0 + \beta_0 + u_8(1-\alpha_1) + u_8(1-\alpha_2))(1-\alpha_3) \\
u_3 &= 4(u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + 4\beta_1 + (2u_8 + 2\alpha_0 + 2\beta_0 + (2u_8 \\
&\quad +\alpha_0)(1-\alpha_1))(1-\alpha_2) + 8u_8 + 8\alpha_0 + 8\beta_0 + 4(2u_8 + \alpha_0)(1-\alpha_1) \\
&\quad +2(5u_8 + 3\alpha_0 + \beta_0 + u_8(1-\alpha_1))(1-\alpha_2) \\
&\quad +(12u_8 + 8\alpha_0 + 4\beta_0 + (2u_8 + \alpha_0)(1-\alpha_1) \\
&\quad +2u_8(1-\alpha_1) + (4u_8 + \alpha_0)(1-\alpha_2))(1-\alpha_3) \\
u_2 &= ((u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + \beta_1)(1-\alpha_2) + \beta_2 \\
&\quad +4(u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + 4\beta_1 \\
&\quad +2(2u_8 + 2\alpha_0 + 2\beta_0 + (2u_8 + \alpha_0)(1-\alpha_1))(1-\alpha_2) \\
&\quad +((u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + \beta_1 + 4u_8 + 4\alpha_0 + 4\beta_0 \\
&\quad +2(2u_8 + \alpha_0)(1-\alpha_1) \\
&\quad +(5u_8 + 3\alpha_0 + \beta_0 + u_8(1-\alpha_1))(1-\alpha_2))(1-\alpha_3) \\
u_1 &= 2((u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + \beta_1)(1-\alpha_2) + 2\beta_2 \\
&\quad +(2(u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + 2\beta_1 \\
&\quad +(2u_8 + 2\alpha_0 + 2\beta_0 + (2u_8 + \alpha_0)(1-\alpha_1))(1-\alpha_2))(1-\alpha_3) \\
u_0 &= (((u_8 + \alpha_0 + \beta_0)(1-\alpha_1) + \beta_1)(1-\alpha_2) + \beta_2)(1-\alpha_3) + \beta_3.
\end{aligned}\right.$$

Computing the coefficients $c, \alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_m$ and $\beta_0, \beta_1, \beta_2, \ldots, \beta_m$ is rather complicated. In practice, it may be a long trial-and-error process (most values of c will give inconvenient solutions), but this is done once and for all. For instance, if $n = 7$, $u_i = 1/i$ for $i \geq 1$, $u_0 = 1$, and $c = 1$, there are several solutions to the system of equations. One of them is

$$\begin{aligned}
\alpha_0 &= -0.85714285714286 \\
\alpha_1 &= -1.01861477121502 \\
\alpha_2 &= 0 \\
\alpha_3 &= -4.58138522878498 \\
\beta_0 &= 1.96666666666667 \\
\beta_1 &= -6.09666666666667 \\
\beta_2 &= 20.7534008337147 \\
\beta_3 &= -94.7138478361582.
\end{aligned}$$

In this case, the transformation allows us to evaluate the polynomial using six multiplications, instead of eight with Horner's scheme. I am not aware of any use of that method for implementing

elementary functions in floating-point arithmetic. This is probably due to the fact that in practice, there is always some parallelism available, so that the simpler methods presented in next section become preferable. Also, the preprocessing of the coefficients may of course introduce some accuracy loss. Another polynomial evaluation algorithm based on some preprocessing of the coefficients was introduced by Paterson and Stockmeyer [379].

5.2 Evaluating Polynomials When Some Parallelism is Available

5.2.1 Generalizations of Horner's Scheme

A first parallel solution is the *second order Horner's scheme*, that consists in splitting up the polynomial into its odd and even parts. Consider for instance the evaluation of

$$p(x) = a_7x^7 + a_6x^6 + \cdots + a_0.$$

One can first evaluate $y = x^2$, then (using Horner's scheme), in parallel:

$$p_{\text{even}} = ((a_6y + a_4) \cdot y + a_2) \cdot y + a_0,$$

and

$$p_{\text{odd}} = ((a_7y + a_5) \cdot y + a_3) \cdot y + a_1,$$

and obtain the final result as

$$p(x) = p_{\text{even}} + x \cdot p_{\text{odd}}.$$

This idea was generalized by Dorn [158]. Assume we wish to evaluate $a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$. Let k be an integer between 2 and $n/2$. Dorn's method consists in precalculating $\alpha = x^k$, and then (possibly using Horner's scheme), in computing in parallel

$$\begin{array}{lcl}
b_0 & = & a_0 + a_k\alpha + a_{2k}\alpha^2 + a_{3k}\alpha^3 + \cdots, \\
b_1 & = & a_1 + a_{k+1}\alpha + a_{2k+1}\alpha^2 + a_{3k+1}\alpha^3 + \cdots, \\
b_2 & = & a_2 + a_{k+2}\alpha + a_{2k+2}\alpha^2 + a_{3k+2}\alpha^3 + \cdots, \\
\cdots & \cdots & \cdots \\
b_{k-1} & = & a_{k-1} + a_{2k-1}\alpha + a_{3k-1}\alpha^2 + a_{4k-1}\alpha^3 + \cdots.
\end{array}$$

We then obtain $p(x)$ as

$$p(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{k-1}x^{k-1}.$$

Another way of parallelizing the evaluation of a polynomial was due to Estrin. Let us introduce it now.

5.2.2 Estrin's Method

Assume that we want to evaluate a degree-7 polynomial:

$$a_7x^7 + a_6x^6 + \cdots + a_1x + a_0.$$

If we are able to perform multiplications and accumulations in parallel (or in a pipelined fashion), we can use Estrin's algorithm [184, 275], easy to generalize to degrees higher than 7 and especially regular for degrees of the form $2^k - 1$:

Algorithm 14 Estrin's algorithm

- input values: $a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0$, and x,
- output value: $p(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0$.

Perform the following steps:

1. In parallel, compute $X^{(1)} = x^2$, $a_3^{(1)} = a_7x + a_6$, $a_2^{(1)} = a_5x + a_4$, $a_1^{(1)} = a_3x + a_2$, and $a_0^{(1)} = a_1x + a_0$,
2. in parallel, compute $X^{(2)} = \left(X^{(1)}\right)^2$, $a_1^{(2)} = a_3^{(1)}X^{(1)} + a_2^{(1)}$, and $a_0^{(2)} = a_1^{(1)}X^{(1)} + a_0^{(1)}$,
3. compute $p(x) = a_1^{(2)}X^{(2)} + a_0^{(2)}$.

Estrin's method was used for instance in some of INTEL's elementary function programs for the Itanium [115] (in fact, the authors used an exhaustive testing strategy to find the evaluation scheme that was optimal for the various polynomials they had to evaluate and their processor, and it sometimes turned out that the best scheme was Estrin's method).

Assume for instance that we use FMA operators with a 5-cycle latency (which was the case on the Itanium Processor). Table 5.1 shows that, using Estrin's algorithm, we can evaluate a degree-7 polynomial in 19 cycles using one operator, and Table 5.2 shows that with two operators this can be done in 17 cycles.

Notice that Estrin's method can also be used in hardware: the basic idea behind the *Polynomier* [167] consists of performing the multiplications and accumulations required by Estrin's method in pipeline, using a modified Braun's multiplier [53, 244].

5.2.3 Evaluating Polynomials on Modern Processors

In Sections 5.2.1 and 5.2.2, we have studied classical methods for evaluating polynomials when some parallelism is available. In practice, these methods will not always give the best possible solution: we need to adapt the evaluation strategy to the particular context (How many arithmetic operators are available? Is there an efficient FMA operator? What is the depth of the pipeline of the arithmetic operators?). Also, a solution that may seem interesting from the point of view of speed may lead, with the particular coefficients of the polynomial being evaluated and the range of the input value, to an unacceptable evaluation error. From that point of view, the possible availability of a larger precision without delay penalty (such as the "double-extended" precision that was available on Intel X86 architectures) is an important point: as mentioned in [224], the extra precision implies that the order of evaluation of the polynomial becomes much less important as far as accuracy is concerned, so that we can fully utilize the available parallelism, and focus on evaluation delay only. However, in general, there is no fast way of deciding which evaluation strategy leads to the best compromise delay versus accuracy. We need to compare various different evaluation schemes.

Table 5.1 *The scheduling of the various operations involved by the evaluation of* $p(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0$ *on a* 5*-cycle pipelined FMA operator, such as one of those available on the Itanium processor [115].*

At cycle	we start computing	we obtain
0	x^2	–
1	$a_3^{(1)} = a_7x + a_6$	–
2	$a_2^{(1)} = a_5x + a_4$	–
3	$a_1^{(1)} = a_3x + a_2$	–
4	$a_0^{(1)} = a_1x + a_0$	–
5	$x^4 = x^2 \cdot x^2$	x^2
6	–	$a_3^{(1)} = a_7x + a_6$
7	$a_1^{(2)} = a_3^{(1)}x^2 + a_2^{(1)}$	$a_2^{(1)} = a_5x + a_4$
8	–	$a_1^{(1)} = a_3x + a_2$
9	$a_0^{(2)} = a_1^{(1)}x^2 + a_0^{(1)}$	$a_0^{(1)} = a_1x + a_0$
10	–	x^4
11	–	–
12	–	$a_1^{(2)} = a_7x^3 + a_6x^2 + a_5x + a_4$
13	–	–
14	$p(x) = a_1^{(2)}x^4 + a_0^{(2)}$	$a_0^{(2)} = a_3x^3 + a_2x^2 + a_1x + a_0$
15	–	–
16	–	–
17	–	–
18	–	–
19	–	$p(x) = a_7x^7 + a_6x^6 + a_5x^5$ $+a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

Table 5.2 *The scheduling of the various operations involved by the evaluation of* $p(x) = a_7x^7 + a_6x^6 + \cdots + a_1x + a_0$ *with* two 5*-cycle pipelined FMA operators, such as the ones available on the Itanium processor [115].*

At cycle	we start computing (operator 1)	we start computing (operator 2)	we obtain (operator 1)	we obtain (operator 2)
0	x^2	$a_3^{(1)} = a_7x + a_6$	–	–
1	$a_2^{(1)} = a_5x + a_4$	$a_1^{(1)} = a_3x + a_2$	–	–
2	$a_0^{(1)} = a_1x + a_0$	–	–	–
3	–	–	–	–
4	–	–	–	–
5	$x^4 = x^2 \cdot x^2$	–	x^2	$a_3^{(1)}$
6	$a_1^{(2)} = a_3^{(1)}x^2 + a_2^{(1)}$	–	$a_2^{(1)}$	$a_1^{(1)}$
7	$a_0^{(2)} = a_1^{(1)}x^2 + a_0^{(1)}$	–	$a_0^{(1)}$	–
8	–	–	–	–
9	–	–	–	–
10	–	–	x^4	–
11	–	–	$a_1^{(2)}$	–
12	$p(x) = a_1^{(2)}x^4 + a_0^{(2)}$	–	$a_0^{(2)}$	–
13	–	–	–	–
14	–	–	–	–
15	–	–	–	–
16	–	–	–	–
17	–	–	$p(x)$	–

When the degree of the polynomial is small, and if we restrict somehow the space of the possible evaluation schemes (for instance by only allowing FMAs), a brute force strategy (comparison of all possible evaluation schemes) is possible. This was done for polynomial evaluation on the INTEL Itanium [224]. Unfortunately, when the degree of the polynomial becomes large, the number of evaluation schemes increases drastically. Mouilleron and Révy [351] found 1304066578 possible evaluation schemes for a degree-6 polynomial. And the situation will probably not improve in the forthcoming years: first, we tend to use polynomials of larger and larger degree (to limit memory access), and second, we may want[4] to be able to chose the evaluation scheme at compile time, which prevents from doing exhaustive or almost-exhaustive search.

The CGPE tool[5] was designed by Révy, Mouilleron and Najahi in order to help in synthesizing fast and certified codes for the evaluation of polynomials, that are optimized for a specific target architecture. Currently, it is adapted to fixed-point arithmetic, but the techniques used by the authors (partly described in [351]) could readily be adapted to floating-point arithmetic. The idea behind the tool is to find good heuristics to reduce the number of evaluation schemes that are compared. CGPE was used successfuly for generating a significant part of the code of the FLIP (*Floating-point Library for Integer Processors*) library [36, 255].[6]

5.3 Computing Bounds on the Evaluation Error

In Chapters 3 and 4, we have focused on the maximum error obtained when approximating a function by a polynomial. Another error must be taken into account: when the polynomial approximation is evaluated in finite precision arithmetic, rounding errors will occur at (almost) each arithmetic operation, resulting in a global evaluation error. We therefore have to find sharp bounds on that polynomial evaluation error, which is the goal of this section.

Once we know the polynomial evaluation error, combining the approximation error and the polynomial evaluation error is done as follows. Assume that f is the function being approximated, P is the approximating polynomial, and $\hat{P}(x)$ is the computed (i.e., with rounding errors) value of P at point x.

- if we discuss in terms of *absolute* error bounds, if ϵ_1 is the bound on the approximation error and ϵ_2 is the bound on the polynomial evaluation error, then for any x, $|f(x) - P(x)| \leq \epsilon_1$ and $|P(x) - \hat{P}(x)| \leq \epsilon_2$, so that the final absolute error satisfies

$$|f(x) - \hat{P}(x)| \leq \epsilon_1 + \epsilon_2;$$

- if we discuss in terms of *relative* error bounds if ϵ_1 is the bound on the approximation error and ϵ_2 is the bound on the polynomial evaluation error, then for any x, $|f(x) - P(x)|/|f(x)| \leq \epsilon_1$ and $|P(x) - \hat{P}(x)|/|P(x)] \leq \epsilon_2$, so that the final relative error satisfies

$$\frac{|f(x) - \hat{P}(x)|}{|f(x)|} \leq \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2.$$

[4] At the time I am writing these lines, we are still far from this goal.

[5] Available at http://cgpe.gforge.inria.fr/.

[6] FLIP is available at http://flip.gforge.inria.fr.

This last value is of course very close to $\epsilon_1 + \epsilon_2$ in all practical cases, and yet, if one wishes to provide certain results, one should not blindly add relative errors.

In the following, we assume that the computations are performed using binary, precision-p, floating-point arithmetic.

Let

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

be a degree-n polynomial. We assume that the a_i's are exactly representable in the floating-point format being used. We wish to tightly bound the largest possible evaluation error, for $x \in [x_{min}, x_{max}]$.

5.3.1 Evaluation Error Assuming Horner's Scheme is Used

If P is evaluated using Horner's scheme, the classical result is the following (see Higham's book [232] for more information):

Theorem 14 *Assume radix-2 and precision-p, rounded to nearest, floating-point arithmetic. The absolute error in the evaluation of*

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using Horner's scheme (with separate additions and multiplications, i.e., no FMAs) is bounded by

$$\gamma_{2n} \cdot \sum_{i=0}^{n} |a_i| \cdot |x|^i,$$

where γ_k is defined as $k \cdot 2^{-p}/(1 - k \cdot 2^{-p})$.

In a recent paper [402], Rump, Bünger, and Jeannerod improved on this last result, and proved the following theorem:

Theorem 15 (Adaptation to radix-2 arithmetic of Theorem 1.3 of [402]) *Assume radix-2 and precision-p, rounded to nearest, floating-point arithmetic. The absolute error in the evaluation of*

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using Horner's scheme (with separate additions and multiplications, i.e., no FMAs) is bounded by

$$n \cdot 2^{-p+1} \cdot \sum_{i=0}^{n} |a_i| \cdot |x|^i,$$

if $n < (1/2) \cdot \left(2^{p/2} - 1\right)$.

The constraint $n < (1/2) \cdot (2^{p/2} - 1)$ is not too restrictive. The bound given by Theorem 15 is valid for polynomials of degree less than or equal to 2047 in binary32/single precision arithmetic, and less than or equal to 47453132 in binary64/double precision arithmetic: in practice, for approximating elementary functions, we use polynomials of a much smaller degree.

This last result might suffice when a very tight bound is not needed (for instance, when the intermediate calculations are performed with a precision that is significantly larger than the target precision). When a very sharp bound is needed, we must use other techniques such as the one we are going to examine now. It can of course be used for computing error bounds by paper and pencil calculations. And yet, that would be so tedious and error-prone, that I strongly recommend that these calculations be automated[7]. Below, I give small and simple Maple programs for performing them. These programs can be generalized to cases where algorithms such as the Fast2Sum and Fast2Mult algorithms (see Chapter 2) are used—at least with the leading coefficients—to get very accurate results. They use the rounding and ulp functions given in Section 2.2.5. Let us first assume that the polynomial will be evaluated using conventional floating-point additions and multiplications (that is, there is no available fused multiply–add instruction).

Evaluation using floating-point additions and multiplications

We assume that we evaluate $p(x)$ using Horner's rule. We also assume that the basic operations used are floating-point additions and multiplications, and that round-to-nearest rounding function is selected (adaptation to other rounding functions is rather straightforward). Define

$$\begin{cases} P^*[i] & = a_n x^{n-i+1} + a_{n-1}x^{n-i} + \cdots + a_i x \\ S^*[i-1] & = a_n x^{n-i+1} + a_{n-1}x^{n-i} + \cdots + a_i x + a_{i-1}. \end{cases}$$

These variables denote the "exact" values that would be successively computed (from $i = n$ to 0), during Horner's evaluation of $p(x)$, for a given $x \in [x_{min}, x_{max}]$ if there were no rounding errors. The exact value of $p(x)$ is $S^*[0]$. We will also denote $P[i]$ and $S[i]$ the *computed* values of $P^*[i]$ and $S^*[i]$, using the relations

$$\begin{cases} P[i-1] = \mathrm{RN}(S[i-1] \cdot x) \\ S[i-1] = \mathrm{RN}(P[i] + a_{i-1}), \end{cases}$$

with $S[n] = a_n$. The computed value of $p(x)$ is $S[0]$. We are going to build lower bounds $P_{min}[i]$ and $S_{min}[i]$, and upper bounds $P_{max}[i]$ and $S_{max}[i]$, on $P[i]$ and $S[i]$. These bounds, of course, will hold for any $x \in [x_{min}, x_{max}]$. To compute them, we will need other variables: $\hat{P}_{min}[i]$ and $\hat{P}_{max}[i]$ will bound the exact value of $S[i]x$, and $\hat{S}_{min}[i-1]$ and $\hat{S}_{max}[i-1]$ will bound the exact value of $P[i] + a_{i-1}$.

To compute an upper bound on the error occurring when evaluating $p(x)$ in floating-point arithmetic, we will need to evaluate the following intermediate error bounds:

- $\delta[i]$ is an upper bound on the error due to the floating-point multiplication of $S[i]$ by x;
- $\epsilon[i-1]$ is an upper bound on the error due to the floating-point addition of $P[i]$ and a_{i-1}.

Now, define $\mathrm{err}[i]$ as an upper bound on $|S^*[i] - S[i]|$. We wish to compute the value of $\mathrm{err}[0]$: this is the bound on the final evaluation error we are looking for. We will compute it iteratively, starting from $\mathrm{err}[n] = 0$.

We first start from the straightforward values $S_{min}[n] = S_{max}[n] = a_n$, and we define $\mathrm{err}[n] = 0$. Now, assume that we know $S_{min}[i]$, $S_{max}[i]$, and $\mathrm{err}[i]$. Let us see how to deduce $S_{min}[i-1]$, $S_{max}[i-1]$, and $\mathrm{err}[i-1]$. First, we obviously find (this is the usual interval multiplication)

$$\hat{P}_{min}[i] = \min\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\},$$

[7] As a matter of fact, similar calculations are done by the Gappa software, presented in Section 2.2.4, so in practice the best solution is to use Gappa.

and

$$\hat{P}_{max}[i] = \max\left\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\right\}.$$

Since we use correctly rounded multiplication, in round-to-nearest mode, the rounding error that occurs when computing $P[i] = \mathrm{RN}(S[i] \cdot x)$ is upper bounded by

$$\frac{1}{2}\,\mathrm{ulp}\,(S[i]x).$$

Since $\mathrm{ulp}\,(t)$ is an increasing function of $|t|$, and since $S[i]x \in [\hat{P}_{min}[i],\ \hat{P}_{max}[i]]$, we get the following bound on that rounding error

$$\delta[i] = \frac{1}{2}\,\mathrm{ulp}\,\left(\max\left\{|\hat{P}_{min}[i]|, |\hat{P}_{max}[i]|\right\}\right).$$

From

$$\hat{P}_{min}[i] \leq S[i]x \leq \hat{P}_{max}[i]$$

and the monotonicity of the RN function, we deduce

$$\mathrm{RN}\left(\hat{P}_{min}[i]\right) \leq P[i] = \mathrm{RN}\,(S[i]x) \leq \mathrm{RN}\left(\hat{P}_{max}[i]\right),$$

which leads us to the following choices for the lower and upper bounds on $P[i]$:

$$\begin{cases} P_{min}[i] = \mathrm{RN}\left(\hat{P}_{min}[i]\right) \\ P_{max}[i] = \mathrm{RN}\left(\hat{P}_{max}[i]\right). \end{cases}$$

Similarly, we find

$$\hat{S}_{min}[i-1] = P_{min}[i] + a_{i-1}$$

and

$$\hat{S}_{max}[i-1] = P_{max}[i] + a_{i-1}.$$

From these values, we deduce a bound on the error that occurs when computing $S[i-1] = \mathrm{RN}(P[i] + a_{i-1})$:

$$\epsilon[i-1] = \frac{1}{2}\,\mathrm{ulp}\,\left(\max\left\{|\hat{S}_{min}[i-1]|, |\hat{S}_{max}[i-1]|\right\}\right),$$

We also find the following lower and upper bounds on $S[i-1]$:

$$\begin{cases} S_{min}[i-1] = \mathrm{RN}\left(\hat{S}_{min}[i-1]\right) \\ S_{max}[i-1] = \mathrm{RN}\left(\hat{S}_{max}[i-1]\right). \end{cases}$$

We now have all the information we need to choose an adequate value for $\mathrm{err}\,[i-1]$. We have,

$$\begin{aligned}|S^*[i-1]-S[i-1]| &= |P^*[i]+a_{i-1}-S[i-1]|\\ &= |S^*[i]x+a_{i-1}-S[i-1]|\\ &\le |S^*[i]-S[i]|\cdot|x|+|S[i]x+a_{i-1}-S[i-1]|\\ &\le \mathrm{err}\,[i]\cdot|x|+|S[i]x-P[i]|+|P[i]+a_{i-1}-S[i-1]|\,,\end{aligned}$$

which immediately leads us to the following choice:

$$\mathrm{err}\,[i-1] = \mathrm{err}\,[i]\max\{|x_{min}|,|x_{max}|\}+\delta[i]+\epsilon[i-1].$$

The following Maple program uses these relations for deducing $\mathrm{err}\,[0]$ from an array a containing the coefficients of p ($a[i]$ is the coefficient of degree i), a variable n representing the degree of p, and the bounds x_{min} and x_{max} on x. It uses the functions `nearest_binary32` and `ulp_in_binary_32` presented in Section 2.2.5, hence it computes error bounds assuming the polynomial evaluation will be done in binary32/single precision arithmetic. To adapt it to calculations performed in binary64 arithmetic, it suffices to replace these functions by functions `nearest_binary64` and `ulp_in_binary_64`.

```
Errevalpol := proc(a,n,xmin,xmax);
smin[n]:= a[n];
smax[n]:= a[n];
err[n]:= 0;
for i from n by -1 to 1 do
  pminhat[i] := min(smin[i]*xmin,smin[i]*xmax,smax[i]*xmin,
                    smax[i]*xmax);
  pmaxhat[i] := max(smin[i]*xmin,smin[i]*xmax,smax[i]*xmin,
                    smax[i]*xmax);
  delta[i]:=0.5*ulp_in_binary_32(max(abs(pminhat[i]),abs(pmaxhat[i])));
  pmin[i] := nearest_binary32(pminhat[i]);
  pmax[i] := nearest_binary32(pmaxhat[i]);
  sminhat[i-1] := pmin[i]+a[i-1];
  smaxhat[i-1] := pmax[i]+a[i-1];
  epsilon[i-1] := 0.5*ulp_in_binary_32(max(abs(sminhat[i-1]),
                    abs(smaxhat[i-1])));
  smin[i-1] := nearest_binary32(sminhat[i-1]);
  smax[i-1] := nearest_binary32(smaxhat[i-1]);
  err[i-1] := err[i]*max(abs(xmin),abs(xmax))
                +epsilon[i-1]+delta[i]
 od;
err[0];
end;
```

Let us now give two examples that illustrate this method and compare it with other solutions.

Example 4 *Consider the polynomial*

$$P(x) = 1 + x + \frac{x^2}{2} + \frac{5592383}{2^{25}}\cdot x^3 + \frac{701583}{2^{24}}\cdot x^4.$$

The coefficients of that polynomial are exactly representable in the binary32 format. That polynomial (generated by Sollya) is an approximation to function e^x in the domain $[0, \ln(2)/64]$. Assume one wishes to bound the error committed when evaluating that polynomial in binary32 floating-point arithmetic, using Horner's scheme (without FMAs), for $x \in [2^{-7}, \ln(2)/64]$. Let us compare four different solutions for bounding that error.

Use of Theorem 15 *First, the bound given by Theorem 15 is*

$$4 \cdot 2^{-23} \cdot \left(1 + \rho + \frac{\rho^2}{2} + \frac{5592383}{2^{25}} \cdot \rho^3 + \frac{701583}{2^{24}} \cdot \rho^4\right),$$

with $\rho = \ln(2)/64$. *This gives* 4.821×10^{-7}.

Use of our method *Using the* `Errevalpol` *Maple program we have just given, we obtain a significantly smaller bound:* 6.073×10^{-8}.

Use of Gappa *The Gappa software, presented in Section 2.2.4, can be called with the following input file (that describes what we actually compute and what we want to compute):*

```
@RN = float<ieee_32,ne>;
# defines RN as round-to-nearest in binary32 arithmetic

x = RN(xx);
a0 = 1;
a1 = 1;
a2 = 1/2;
a3 = 5592383b-25;
a4 = 701583b-24;

# description of the program

p4 = RN(a4*x);
s3 = RN(a3+p4);
p3 = RN(s3*x);
s2 = RN(a2+p3);
p2 = RN(s2*x);
s1 = RN(a1+p2);
p1 = RN(s1*x);
s0 = RN(a0+p1);

# description of the exact value we are approximating
# convention: an "m" as a prefix of the names of "exact" variables
# mp4, ms3, mp3, ms2, , ms0 are just here
# to help describing the hint to Gappa

mp4 = (a4*x);
ms3 = (a3+mp4);
mp3 = (ms3*x);
ms2 = (a2+mp3);
mp2 = (ms2*x);
ms1 = (a1+mp2);
mp1 = (ms1*x);
ms0 = (a0+mp1);
mpolynomial = a4*x*x*x*x + a3*x*x*x + a2 *x*x + a1*x + a0;
epsilon = (mpolynomial-s0);

# description of what we want to prove

{
# input hypothesis
x in [1b-7,1453635b-27]

->
# goal to prove
```

```
|epsilon| in ?
}

# hint to Gappa
mpolynomial -> ms0 ;
```

Notice that to get a relative error bound, it would have sufficed to replace the line

```
epsilon = (mpolynomial-s0);
```

by the line

```
epsilon = (mpolynomial-s0)/mpolynomial;
```

Called with that input file, Gappa will return

```
|epsilon| in [0, 293650037469320933b-82
                  {6.07254e-08, 2^(-23.9731)}]
```

Hence the bound given by Gappa is the same as the one we obtained with our method: 6.073×10^{-8}. *This is not surprising: it is very likely that Gappa uses a similar strategy for computing the error bound. An important advantage of Gappa is that if called with option* `-Bcoq` *it generates a formal proof of the bound, that can be checked with the Coq proof checker.*

Exhaustive testing *There are only 3240472 binary32 numbers between* 2^{-7} *and* $\ln(2)/64$*: it takes a few minutes only to evaluate the polynomial in binary32 arithmetic and exactly (using for instance Maple) for all these numbers, and find that the largest attained evaluation error is* 6.06955×10^{-8}.

On this example, Gappa and our `Errevalpol` program give a very tight error bound, and the bound given by Theorem 15, although slightly pessimistic, remains reasonable. Notice that frequently, for floating-point formats not wider than binary32, performing exhaustive tests is a sound solution. Let us now switch to another example.

Example 5 *Consider, for* $x \in [1/2, 1]$*, and assuming binary64/double precision arithmetic, the polynomial*

$$q(x) = \frac{4502715367124429}{2^{52}} - \frac{5094120834338589}{2^{53}}\,x + \frac{3943097548915637}{2^{52}}\,x^2 - \frac{272563672039763}{2^{49}}\,x^3 + \frac{6289926120511169}{2^{55}}\,x^4.$$

It is an (rather poor, but this does not matter here) approximation to $x! = \Gamma(x+1)$*. The bound provided by Theorem 15 is* 1.479×10^{-6}*. Using function* `Errevalpol` *(adapted to the binary64 format), we get a much smaller error bound, equal to* 3.4695×10^{-16}*, the bound given by Gappa is the same. An exhaustive test is not possible in binary64 arithmetic, however, the largest error we have actually obtained through experiments is around* 2.546×10^{-16}*, which is significantly less than the computed bounds.*

This large difference comes from a well-known problem in interval arithmetic. The bounds $\hat{P}_{min}[i]$ and $\hat{P}_{max}[i]$ are obtained by an interval multiplication of $[S_{min}[i], S_{max}[i]]$ by $[x_{min}, x_{max}]$. For getting the bounds of that interval product, depending on i, it is sometimes x_{min} that is used, and sometimes x_{max}: we lose the essential information that, in actual polynomial evaluations, it is "the same x" that

is used at all steps of Horner's method. This problem did not occur in Example 4, because, since the polynomial coefficients were all positive, and since $x_{min} \geq 0$, it was always x_{min} that was used to get the lower bounds $\hat{P}_{min}[i]$, and it was always x_{max} that was used to get the upper bounds $\hat{P}_{max}[i]$.

The best way to make that problem negligible is to split the input interval $[x_{min}, x_{max}]$ into several subintervals, to use `Errevalpol` in each subdomain, and to consider the largest returned error bound. This is done by the following Maple program:

```
RefinedErrPol := proc(a,n,xmin,xmax,NumbOfIntervals);
errmax := 0;
Size := (xmax-xmin)/NumbOfIntervals;
for i from 0 to NumbOfIntervals-1 do
 err := Errevalpol(a,n,xmin+i*Size,xmin+(i+1)*Size);
 if err > errmax then errmax := err fi
od;
errmax
end;
```

In our example, if we cut the initial input interval into 128 subintervals, by calling `RefinedErrPol (a,4,1/2,1,128)`, we get a better error bound: 2.919×10^{-16}. Gappa does the domain splitting in 128 subintervals if, at the end of the input file, we add the hint

```
$ x in 128;
```

and we then obtain the bound 2.9165×10^{-16}. If we ask Gappa to split the input domain into 1024 subintervals, it takes a few seconds to obtain an even better bound: 2.91494×10^{-16}.

Evaluation using fused multiply–accumulate instructions

Let us now assume that on the target architecture a fused multiply–accumulate (FMA) instruction is available, and that we use that instruction to implement the polynomial evaluation by Horner's scheme.

As previously, we define

$$S^*[i] = a_n x^{n-i} + a_{n-1} x^{n-i-1} + \cdots + a_i,$$

we also define $S[i]$ as the computed value, for a given x, of $S^*[i]$ obtained by iteratively using

$$S[i-1] = \mathrm{RN}\left(S[i]x + a_{i-1}\right),$$

with $S[n] = a_n$. We will compute lower and upper bounds $S_{min}[i]$ and $S_{max}[i]$ on $S[i]$. To do that, we use intermediate variables $\hat{S}_{min}[i-1]$ and $\hat{S}_{max}[i-1]$ that bound the exact value of $(S[i]x + a_{i-1})$, and a variable $\epsilon[i]$ that bounds the rounding error occurring when computing $S[i]$ from $S[i+1]$.

As in Section 5.3.1, $\mathrm{err}[i]$ is an upper bound on $|S^*[i] - S[i]|$. We wish to compute $\mathrm{err}[0]$: this is the bound we are looking for on the final evaluation error. We will compute it iteratively, starting from $\mathrm{err}[n] = 0$. The iterative process that gives $\mathrm{err}[0]$ is very similar to the one described in Section 5.3.1. We first start from the straightforward values: $S_{min}[n] = S_{max}[n] = a_n$ and $\mathrm{err}[n] = 0$.

Now, assume that we know $S_{min}[i]$, $S_{max}[i]$ and $\mathrm{err}[i]$. Let us see how to deduce $S_{min}[i-1]$, $S_{max}[i-1]$ and $\mathrm{err}[i-1]$. We find

$$\hat{S}_{min}[i-1] = a_{i-1} + \min\left\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\right\}$$

and

$$\hat{S}_{max}[i-1] = a_{i-1} + \max\left\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\right\}.$$

We then deduce

$$\epsilon[i-1] = \frac{1}{2}\,\mathrm{ulp}\left(\max\left\{|\hat{S}_{min}[i-1]|, |\hat{S}_{max}[i-1]|\right\}\right).$$

Also, since RN is a monotonic function, from $\hat{S}_{min}[i-1] \leq a_{i-1} + S[i]x \leq \hat{S}_{max}[i-1]$, we deduce

$$\mathrm{RN}\left(\hat{S}_{min}[i-1]\right) \leq \mathrm{RN}\left(a_{i-1} + S[i]x\right) = S[i-1] \leq \mathrm{RN}\left(\hat{S}_{max}[i-1]\right).$$

Hence, a natural choice is

$$\begin{cases} S_{min}[i-1] = \mathrm{RN}\left(\hat{S}_{min}[i-1]\right) \\ S_{max}[i-1] = \mathrm{RN}\left(\hat{S}_{max}[i-1]\right). \end{cases}$$

We now have all the information we need to choose an adequate value for $\mathrm{err}\,[i-1]$:

$$\mathrm{err}\,[i-1] = \mathrm{err}\,[i] \max\{|x_{min}|, |x_{max}|\} + \epsilon[i-1].$$

The following Maple program uses these relations for deducing $\mathrm{err}\,[0]$ from an array a containing the coefficients of p, a variable n representing the degree of p, and the bounds x_{min} and x_{max} on x. It uses the functions `nearest_binary32` and `ulp_in_binary_32` presented in Section 2.2.5, hence it computes error bounds assuming the polynomial evaluation will be done in binary32/single precision arithmetic. To adapt it to calculations performed in binary64 arithmetic, it suffices to replace these functions by functions `nearest_binary64` and `ulp_in_binary_64`.

```
ErrevalpolFMA := proc(a,n,xmin,xmax);
smin[n]:= a[n];
smax[n]:= a[n];
err[n]:= 0;
for i from n by -1 to 1 do
  sminhat[i-1] := a[i-1] + min(smin[i]*xmin,smin[i]*xmax,
                  smax[i]*xmin,smax[i]*xmax);
  smaxhat[i-1] := a[i-1] + max(smin[i]*xmin,smin[i]*xmax,
                  smax[i]*xmin,smax[i]*xmax);
  epsilon[i-1] := 0.5*ulp_in_binary_32(max(abs(sminhat[i-1]),
                  abs(smaxhat[i-1])));
  smin[i-1] := nearest_binary32(sminhat[i-1]);
  smax[i-1] := nearest_binary32(smaxhat[i-1]);
  err[i-1] := err[i]*max(abs(xmin),abs(xmax))+epsilon[i-1]
 od;
err[0];
end;
```

Example 6 • *With the polynomial of Example 4, the Maple program* `ErrevalpolFMA` *and Gappa return the same bound* 6.026×10^{-8}. *An exhaustive testing shows that the largest attained error is* 6.021×10^{-8}.

- *With the polynomial of Example 5, the Maple program* `ErrevalpolFMA` *(adapted to binary64 calculations) returns* 2.2205×10^{-16} *without splitting of the input domain, and* 1.9451×10^{-16} *if we split the input domain into 128 parts. Gappa gives the same figures. Exhaustive testing is out of reach, but the largest error we have encountered in our experiments was* 1.7787×10^{-16}.

These examples show again that we can obtain very tight bounds on the evaluation error. Incidentally, these examples also illustrate something that frequently happens. The use of an FMA operator allows one to halve the number of operations needed to perform the polynomial evaluation: one could expect

significant gains in terms of delay and accuracy. The gains in terms of delay are here, and yet the gains in terms of accuracy, although not negligible, are rather small.

5.3.2 Evaluation Error with Methods Different than Horner's Scheme

As we have seen before, as soon as its degree is large, the number of possible evaluation strategies for a given polynomial is huge. We cannot try to find and prove a theoretical result such as Theorem 15 for each of these evaluation strategies. It is possible to build ad hoc programs similar to the Maple programs of the previous section for computing error bounds for a given strategy. In general, one should avoid that (the only possible exception being the case when we know we will use the same strategy for many functions), because it is a tedious and error-prone process. Gappa, with its versatility, its ability to generate a formal proof, and with the useful fact that it can be called from other tools is certainly the best current solution. Consider, for example, a degree-4 polynomial

$$P(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0.$$

If enough parallelism is available and there is an FMA instruction, one may consider evaluating it as follows:

Algorithm 15 Evaluation strategy for a degree-4 polynomial, that exhibits some parallelism

$y \leftarrow \mathrm{RN}(x^2)$
$b_1 \leftarrow \mathrm{RN}(a_4x + a_3)$
$b_0 \leftarrow \mathrm{RN}(a_2x + a_1)$
$c \leftarrow \mathrm{RN}(b_1y + b_0)$
$s \leftarrow \mathrm{RN}(cx + a_0)$

the advantage being that the first three lines can be executed in parallel. A very simple modification of the Gappa input file we have presented in Example 4 allows to deal with this evaluation scheme. For instance, the polynomial of Example 4, with inputs in $[2^{-7}, \ln(2)/64]$, is dealt with the following Gappa input file.

```
@RN = float<ieee_32,ne>;
# defines RN as round-to-nearest in binary32 arithmetic

x = RN(xx);
a0 = 1;
a1 = 1;
a2 = 1/2;
a3 = 5592383b-25;
a4 = 701583b-24;

# description of the program

y =  RN(x*x);
b1 = RN(a4*x+a3);
b0 = RN(a2*x+a1);
c = RN(b1*y+b0);
s = RN(c*x+a0);

# description of the exact value we are approximating
```

```
# convention: an "M" as a prefix of the names of "exact" variables

My =  (x*x);
Mb1 = (a4*x+a3);
Mb0 = (a2*x+a1);
Mc = (Mb1*My+Mb0);
Ms = (Mc*x+a0);
Mpolynomial = a4*x*x*x*x + a3*x*x*x + a2*x*x + a1*x + a0;
epsilon = (Mpolynomial-s);

# description of what we want to prove

{
# input hypothesis
x in [1b-7,1453635b-27]

->
# goal to prove
|epsilon| in ?

}

# hint to Gappa

Mpolynomial -> Ms;
```

The returned bound is 6.08958×10^{-8}, which is close to the error bounds of Horner's scheme (with or without FMA) with the same polynomial. The actual largest error, obtained through exhaustive calculations is around 6.0678×10^{-8}.

If we use the same evaluation scheme with the polynomial and input domain of Example 5, the obtained error bound (with a splitting into 128 subintervals) is 1.8417×10^{-16}. Exhaustive tests are out of reach, but the largest error found in large simulations was 1.6695×10^{-16}.

Incidentally, this shows that general prediction on the numerical quality of evaluation strategies is a difficult exercise: with the first polynomial, Algorithm 15 is (very slightly) less accurate than the FMA version of Horner's algorithm, and with the second polynomial it seems more accurate (although we have no certainty: the only sure fact is that the bounds are lower).

5.3.3 When High Accuracy is Needed

We frequently need to evaluate a polynomial with a precision somehow higher than the target precision. This typically occurs when the result of the polynomial evaluation is not the final result to be returned (for instance, when we have performed a range reduction and we need to deduce the value of the function at the initial argument from its value at the reduced argument), or when we want to guarantee correct rounding and we need to make sure that the value of the function is far enough from a point where the rounding function changes (see Chapter 12). Fortunately, in many cases, it is rather easy to obtain that extra precision. Consider a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \tag{5.2}$$

it frequently happens, with the particular polynomials used for approximating the elementary functions, that the "head" in (5.2), say a_0 or $a_0 + a_1 x$, has a large absolute value compared to the "tail" $a_1 x + \cdots +$

$a_n x^n$ or $a_2 x^2 + \cdots + a_n x^n$, either because the range reduction being used before polynomial evaluation leads to small values of $|x|$, or because the sequence $|a_i|$ quickly decreases. In such cases a simple way to save some extra precision is to evaluate the tail separately, and then to return a double-word result (y_h, y_ℓ), in general obtained by applying the Fast2Sum or the 2Sum algorithm (see Section 2.2.1,) to the pair (head,tail). This double-word result can be used later on to perform further calculations, or to perform a Ziv rounding test (see Section 12.4) to quickly check if we have enough information to return a correctly rounded result.

The following example illustrates this strategy:

Example 7 *The following polynomial, whose coefficients are binary64 numbers, approximates* 2^x*, for* $x \in [0, 1/512]$ *with a relative error less than* 8.387×10^{-20}*:*

$$P(x) = 1 + \frac{6243314768165347}{2^{53}} \cdot x + \frac{8655072058047061}{2^{55}} \cdot x^2 + \frac{3999491778607567}{2^{56}} \cdot x^3 + \frac{5548134412280811}{2^{59}} \cdot x^4.$$

Define a_i *as the degree-i coefficient of* P*. Evaluating* $P(x)$ *in binary64 arithmetic using Horner's scheme with an FMA instruction, i.e., as*

$$\mathrm{RN}(a_0 + x \cdot \mathrm{RN}(a_1 + x \cdot \mathrm{RN}(a_2 + x \cdot \mathrm{RN}(a_3 + x \cdot a_4)))),$$

results in a relative evaluation error less than (but frequently close to) 1.113×10^{-16}*, which is large relative to the approximation error. Now, if we evaluate* $P(x)$ *as follows:*

- *we first evaluate the tail* $a_1 x + \cdots + a_n x^n$ *using Horner's scheme, i.e., as*

$$t = \mathrm{RN}(x \cdot \mathrm{RN}(a_1 + x \cdot \mathrm{RN}(a_2 + x \cdot \mathrm{RN}(a_3 + x \cdot a_4))))$$

- *and we return the double-word*

$$(y_h, y_\ell) = \mathrm{Fast2Sum}(a_0, t),$$

then the relative evaluation error becomes less than (but frequently close to) 2.170×10^{-19}*, which is much better.*

5.4 Polynomial Evaluation by Specific Hardware

When designing or using specific hardware, it may be possible to use some algorithms and architectures for evaluating polynomials that have been proposed in the literature. Let us examine some such solutions.

5.4.1 The E-Method

The *E-method*, introduced by M.D. Ercegovac in [172, 173], allows efficient evaluation of polynomials and certain rational functions on simple and regular hardware. Here we concentrate on the evaluation of polynomials assuming radix-2 arithmetic.

Consider the evaluation of $p(x) = p_n x^n + p_{n-1}x^{n-1} + \cdots + p_0$. One can easily show that $p(x)$ is equal to y_0, where $[y_0, y_1, \ldots, y_n]^t$ is the solution of the following linear system:

$$\begin{bmatrix} 1 & -x & 0 & & \cdots & & 0 \\ 0 & 1 & -x & 0 & & \cdots & 0 \\ 0 & 0 & 1 & -x & 0 & & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & \ddots & 0 \\ & & & & & & 1 & -x \\ 0 & & & \cdots & & & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \\ \vdots \\ \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \\ \vdots \\ \\ p_{n-1} \\ p_n \end{bmatrix}. \tag{5.3}$$

The radix-2 E-method consists of solving this linear system by means of the basic recursion

$$\begin{aligned} w^{(0)} &= [p_0, p_1, \ldots, p_n]^t \\ w^{(j)} &= 2 \times \left[w^{(j-1)} - A d^{(j-1)} \right], \end{aligned} \tag{5.4}$$

where A is the matrix of the linear system. This gives, for $i = 0, \ldots, n$,

$$w_i^{(j)} = 2 \times \left[w_i^{(j-1)} - d_i^{(j-1)} + d_{i+1}^{(j-1)} x \right],$$

where $d_i^{(j)} \in \{-1, 0, 1\}$. Define the number

$$D_i^{(j)} = d_i^{(0)}.d_i^{(1)} d_i^{(2)} \ldots d_i^{(j)}.$$

The $d_i^{(k)}$ are the digits of a radix-2 signed-digit (see Chapter 2) representation of $D_i^{(j)}$. One can show that if for any i the sequences $|w_i^{(j)}|$ are bounded, then $D_i^{(j)}$ goes to y_i as j goes to infinity.

The problem at step j is to find a *selection function* that gives a value of the terms $d_i^{(j)}$ from the terms $w_i^{(j)}$ such that the values $w_i^{(j+1)}$ remain bounded. In [173], the following selection function (a form of rounding) is proposed,

$$s(x) = \begin{cases} \text{sign}\,(x) \times \lfloor |x + 1/2| \rfloor, & \text{if } |x| \leq 1 \\ \text{sign}\,(x) \times \lfloor |x| \rfloor, & \text{otherwise,} \end{cases} \tag{5.5}$$

and applied to the following cases:

1. $d_i^{(j)} = s(w_i^{(j)})$; that is, the selection requires nonredundant $w_i^{(j)}$;
2. $d_i^{(j)} = s(\hat{w}_i^{(j)})$, where $\hat{w}_i^{(j)}$ is an *approximation* of $w_i^{(j)}$. In practice, $\hat{w}_i^{(j)}$ is deduced from a few digits of $w_i^{(j)}$ by means of a rounding to the nearest or a truncation.

Assume

$$\begin{cases} \forall i, |p_i| \leq \xi \\ |x| \leq \alpha \\ |w_i^{(j)} - \hat{w}_i^{(j)}| \leq \frac{\Delta}{2}. \end{cases}$$

The E-method gives a correct result if the previously defined bounds ξ, α, and Δ satisfy

$$\begin{cases} \xi = \frac{1}{2}(1+\Delta) \\ 0 < \Delta < 1 \\ \alpha \leq \frac{1}{4}(1-\Delta). \end{cases} \tag{5.6}$$

For instance, if $\Delta = 1/2$, one can evaluate $p(x)$ for $x \leq 1/8$ and $\max |p_i| \leq 3/4$. Those bounds may seem quite restrictive, but in practice there exist scaling techniques [173] that allow us to compute $p(x)$ for any x and p.

5.4.2 Custom Precision Function Evaluation on Embedded Processors

Lee and Villasenor [304] approximate functions by polynomials that are to be evaluated on fixed-point processors. They use analytical approaches to allow the designer to customize the precision. They evaluate the polynomials using Horner's scheme.

5.4.3 Polynomial Evaluation on FPGAs

Using FPGAs, one can size each arithmetic operator so that it offers the necessary precision and works on the required range, but not more. FloPoCo,[8] coordinated by de Dinechin [137], is a generator of arithmetic cores for FPGAs. There is a significant literature on the implementation on FPGAs of elementary functions through the use of polynomial approximations. Langhammer and Pasca [296] implement the Horner scheme with a "fused" operator (that saves the area and delay usually necessitated by the alignments and normalization that are needed in individual floating-point operations). They also design floating-point polynomial operators dedicated to a restricted input range (the one needed for a given function), which allows for significant savings in terms of operator size [297]. Thomas implements faithfully rounded (see Chapter 12) functions on FPGAs using polynomial approximations [452].

[8] FloPoCo is freely available at http://flopoco.gforge.inria.fr.

Table-Based Methods

6

6.1 Introduction

As explained in Chapter 1, for evaluating a function in a large domain (possibly made up with all floating-point numbers of a given format), we first perform one or several *range reduction* steps, to reduce our initial problem to the evaluation of a function in one or several (sometimes, many!) smaller intervals. In each of these intervals, either the function is approximated by a polynomial whose coefficients are tabulated, or we have to store the value of the function at some point of the interval, and use identities such as $\exp(a+b) = \exp(a) \cdot \exp(b)$ or trigonometric identities to reduce the problem to evaluating the function near zero. Choosing the size of these intervals is not an easy task:

- large intervals will make range reduction simpler and require smaller tables. However, they will require polynomials of large degrees, whose evaluation takes more clock cycles, and for which guaranteeing a tight bound on the evaluation error may be more difficult;
- small intervals have the advantage of requiring very small polynomials quickly evaluated. However, possible cache misses due to the large tables they require may totally destroy performance. See [143, 144] for a discussion on this problem.

To illustrate this, assume one wishes to implement function 2^x in binary64 arithmetic. First, a preliminary reduction to the interval $[0, 1)$ is straightforward: if $k = \lfloor x \rfloor$ and $f = x - k$, then $2^x = 2^f 2^k$, and multiplying by 2^k reduces to a simple exponent manipulation. If one wishes to use polynomial approximations of relative accuracy better than $2^{-54} \approx 5.55 \times 10^{-17}$, and if we add the constraint that the degree-0 coefficient of the polynomial approximation should be 1, and that all the other coefficients should be binary64 numbers, different choices are possible, among them:

- one can use the following degree-11 polynomial that approximates 2^x with relative error at most 3.385×10^{-18} on the whole interval $[0, 1)$:

```
1 + x * (6243314768165365 * 2^(-53)
+ x * (4327536028901397 * 2^(-54)
+ x * (7998985059327887 * 2^(-57)
+ x * (5544473936422889 * 2^(-59)
+ x * (3074509296366713 * 2^(-61)
+ x * (5682892945237327 * 2^(-65)
+ x * (9003886559762327 * 2^(-69)
```


J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_6

```
+ x * (6237463724831189 * 2^(-72)
+ x * (7747239597825951 * 2^(-76)
+ x * (7929123815494859 * 2^(-80)
+ x * (6058874616061415 * 2^(-83))))))))))))
```

- one can have all values $c_i = 2^{i/256}$, $1 \le i \le 255$, stored in a table (possibly as double-word numbers), use the following degree-4 polynomial, that approximates 2^x with relative error at most 2.786×10^{-18} on the interval $[0, 1/256)$, so that if $t = i/256 + \eta$, with $\eta \in [0, 1/256)$, 2^t is computed as $c_i \cdot 2^\eta$:

```
1 + x * (6243314768165159 * 2^(-53)
+ x * (4327536029886647 * 2^(-54)
+ x * (499936188037589 * 2^(-53)
+ x * (5551817238983839 * 2^(-59))))).
```

Depending on the properties of the underlying hardware (size of caches, speed of the arithmetic operations vs speed of memory access, available parallelism…) and on the requirements in terms of accuracy, speed, and memory consumption, the spectrum of chosen compromises ranges from table-only solutions such as the bipartite method, presented in Section 6.5.4 to no-table solutions: for instance the arctangent function described in [115, 224] and designed for the IA-64 architecture uses a polynomial of degree 47 and no table.

Another example that illustrates the link between interval size and polynomial degree is the following.

We wish to approximate function $\ln\left(1 + e^{-x}\right)$ in binary64 arithmetic, in the interval [0, 1], with relative error less than $2^{-54} \approx 5.55 \times 10^{-17}$. We require all coefficients of the polynomial approximation to be binary64 floating-point numbers. The following solutions lead to similar accuracies:

- one polynomial, for the whole domain [0, 1] of degree 14;
- four degree-9 polynomials, for the intervals [0,1/4], [1/4,1/2], [1/2,3/4], and [3/4,1], respectively.

These examples show that a significant amount of polynomial evaluation time can be saved by splitting the original domain. Many compromises between the amount of computation and the amount of memory access can be found. Much care is needed at the boundaries of the subdomains if we wish to preserve properties such as monotonicity. Tables 6.1 and 6.2 show for various functions that reducing the size of the interval where a function is approximated allows the use of a polynomial of small degree. With the simplest functions, it is not necessary to recalculate and store a new polynomial or rational approximation for each subinterval; one can use some simple algebraic properties such as $e^{a+b} = e^a e^b$. For more complex functions such as $\Gamma(x)$ or $\log(1 + e^{-x})$, we need a different approximation for each subinterval.

In this chapter, we study three different classes of table-based methods. The choice among the different methods depends on the kind of implementation (software, hardware) and on the possible availability of a "working precision" (i.e., the precision used for the intermediate calculations) significantly higher than the "target precision" (i.e., the output format):

- first, methods using a "standard table" (i.e., the function is tabulated at regularly spaced values), and a polynomial or rational approximation. To provide last-bit accuracy, those methods must be

Table 6.1 *Absolute error of the minimax polynomial approximations to some functions on the interval* $[0, a]$. *The error decreases rapidly when a becomes small.*

a	arctan, degree 10	exp, degree 4	$\ln(1+x)$, degree 5
5	0.00011	0.83	0.0021
2	1.0×10^{-6}	0.0015	0.00013
1	1.9×10^{-9}	0.000027	8.7×10^{-6}
0.1	3.6×10^{-19}	1.7×10^{-10}	6.1×10^{-11}
0.01	4.3×10^{-30}	1.6×10^{-15}	7.9×10^{-17}

Table 6.2 *Degrees of the minimax polynomial approximations that are required to approximate some functions with absolute error less than* 10^{-5} *on the interval* $[0, a]$. *When a becomes small, a very low degree suffices.*

a	arctan	exp	$\ln(1+x)$
10	19	16	15
1	6	5	5
0.1	3	2	3
0.01	1	1	1

implemented using (or simulating—see Section 2.2.2) a precision that is somewhat larger than the target precision; by simply tabulating the function in the target precision, an error that can be very close to $1/2$ ulp is already committed, so that there is no hope of having the final error bounded by $1/2$ ulp or slightly more than $1/2$ ulp. Due to this problem, such methods are better suited either for hardwired implementation, or when a larger precision is available at reasonable cost. If the target is binary64 precision, this is the case with the x86 instruction set on Intel processors that offer, since the 8087, an internal 80-bit format. Tang's "table-driven" algorithms [447, 448, 449, 450] (see Section 6.2) belong to this class of methods;

- second, methods that use "accurate tables" (i.e., the function is tabulated at *almost* regularly spaced points, for which the value of the function is very close to a machine number). Such methods are attractive for software implementations. With some care, they can be implemented using the target precision only. Gal's "accurate tables method" (see Section 6.3) belongs to this class of methods;
- third, methods using several consecutive lookups in tables and dedicated operators. Examples of such methods are Wong and Goto's algorithms, presented in Section 6.5.1 or the bipartite method, presented in Section 6.5.4. Such methods may be more suited for a hardware implementation.

6.2 Table-Driven Algorithms

In [447, 448, 449, 450], P.T.P. Tang proposes some guidelines for the implementation of the elementary functions using table lookup algorithms. For the computation of $f(x)$, his algorithms use three elementary steps:

reduction: from the input argument x (after a possible preliminary range reduction; see Chapter 11), one deduces a variable y belonging to a very small domain, such that $f(x)$ can easily be deduced from $f(y)$ (or, possibly, from some function $g(y)$). This step can be merged with the preliminary range reduction;

approximation: $f(y)$ (or $g(y)$) is computed using a low-degree polynomial approximation;
reconstruction: $f(x)$ is deduced from $f(y)$ (or $g(y)$).

Now we consider some examples. We first examine in detail the algorithm suggested by Tang for $\exp(x)$ in double-precision IEEE floating-point arithmetic [447]. After this, we briefly give Tang's guidelines for $\ln(x)$ and $\sin(x)$ [449].

6.2.1 Tang's Algorithm for $\exp(x)$ in IEEE Floating-Point Arithmetic

Assume we wish to evaluate $\exp(x)$ in IEEE double-precision floating-point arithmetic.[1] Tang [447] suggests first reducing the input argument to a value r in the interval

$$[-\frac{\ln(2)}{64}, +\frac{\ln(2)}{64}],$$

second to approximate $\exp(r) - 1$ by a polynomial $p(r)$, and finally to reconstruct $\exp(x)$ by the formula

$$\exp(x) = 2^m(2^{j/32} + 2^{j/32}p(r)),$$

where j and m are such that

$$x = (32m + j)\frac{\ln(2)}{32} + r, \quad 0 \le j \le 31. \tag{6.1}$$

These various steps are implemented as follows.

reduction: To make the computation more accurate, Tang represents the reduced argument r as the sum of two floating-point numbers r_1 and r_2 such that $r_2 \ll r_1$ and $r_1 + r_2$ approximates r to a precision higher than the working precision. To do this, Tang uses a method inspired from Cody and Waite's argument reduction algorithm (see Section 11.2). More precisely, he defines three floating-point numbers L^{left}, L^{right} and Λ, such that:

- Λ is $32/\ln(2)$ rounded to double-precision;
- L^{left} has a few trailing zeros;
- $L^{\text{right}} \ll L^{\text{left}}$, and $L^{\text{left}} + L^{\text{right}}$ approximates $\ln(2)/32$ to a precision much higher than the working one.

The numbers r_1 and r_2 are computed as follows. Let N be $x \times \Lambda$ rounded to the nearest integer. Define $N_2 = N \bmod 32$ and $N_1 = N - N_2$. We compute, in the working precision:

$$r_1 = x - N \times L^{\text{left}}$$

[1] For single-precision, see [447].

and

$$r_2 = -N \times L^{\text{right}}.$$

The values m and j of (6.1) are $m = N_1/32$ and $j = N_2$. An analysis of this reduction method is given in reference [186]. Other reduction methods, that may be more accurate for large values of x, are presented in Chapter 11.

approximation: $p(r)$ is computed as follows. First, we compute $r = r_1 + r_2$ in the working precision. Second, we compute

$$Q = r \times r \times (a_1 + r \times (a_2 + r \times (a_3 + r \times (a_4 + r \times a_5)))),$$

where the a_i are the coefficients of a minimax approximation. Finally, we get

$$p(r) = r_1 + (r_2 + Q).$$

The term r_2 is used at order 1 only.

reconstruction: The values $s_j = 2^{j/32}$, $j = 0, \ldots, 31$, are precomputed in higher precision and represented by two double-precision numbers s_j^{left} and s_j^{right} such that:

- $s_j^{\text{left}} \gg s_j^{\text{right}}$;
- the six trailing bits of s_j^{left} are equal to zero;
- $s_j = s_j^{\text{left}} + s_j^{\text{right}}$ to around 100 bits.

Let S_j be the double-precision approximation of s_j. We compute

$$\exp(x) = 2^m \times \left(s_j^{\text{left}} + \left(s_j^{\text{right}} + S_j \times p(r)\right)\right).$$

6.2.2 $\ln(x)$ on $[1, 2]$

reduction: If $x - 1$ is very small (Tang suggests the threshold $e^{1/16}$ for x), then we approximate $\ln x$ by a polynomial. Otherwise, we find "breakpoints" $c_k = 1 + k/64$, $k = 1, 2, \ldots, 64$, such that

$$|x - c_k| \leq \frac{1}{128}.$$

We define $r = 2\,(x - c_k)\,/\,(x + c_k)$. Hence $|r| \leq 1/128$.

approximation: we approximate

$$\ln\left(\frac{x}{c_k}\right) = \ln\left(\frac{1 + r/2}{1 - r/2}\right)$$

for $r \in [0, 1/128]$ by a polynomial $p(r)$ of the variable r. Depending on the required accuracy, one can use one of the polynomials (with binary64 coefficients) given in Table 6.3.

Table 6.3 *Approximations to* $\ln((1+r/2)/(1-r/2))$ *on* $[0, 1/128]$.

Degree	Relative error	Polynomial
3	8.085×10^{-12}	r $+6004845316577347 \times 2^{-56} \cdot r^3$
5	2.005×10^{-17}	r $+6004799502898513 \times 2^{-56} \cdot r^3$ $+900733669546681 \times 2^{-56} \cdot r^5$
7	8.036×10^{-23}	r $+6004799503160663 \times 2^{-56} \cdot r^3$ $+450359962663711 \times 2^{-55} \cdot r^5$ $+5147094262912473 \times 2^{-61} \cdot r^7$
9	1.685×10^{-23}	r $+6004799503160661 \times 2^{-56} \cdot r^3$ $+1801439851004373 \times 2^{-57} \cdot r^5$ $+5146948095108811 \times 2^{-61} \cdot r^7$ $+4731161337446337 \times 2^{-63} \cdot r^9$

reconstruction: we get $\ln(x)$ from

$$\begin{aligned} ln(x) &= \ln(c_k) + \ln(x/c_k) \\ &\approx \ln(c_k) + p(r). \end{aligned}$$

The values $\ln(c_k)$ are tabulated.

If a better accuracy is required, one can use more breakpoints; with 256 breakpoints and a polynomial (with binary64 coefficients, and still with the degree-1 coefficient being 1) of degree 7, the approximation error will be 1.803×10^{-24}. Using 512 breakpoints, the approximation error will be 4.503×10^{-25} with a polynomial of degree 7, and 2.632×10^{-25} with a polynomial of degree 9. The improvement is not tremendous: this is because imposing the coefficients to be binary64 numbers is a significant constraint. Much better approximations are obtained by allowing the degree-3 coefficient to be a double-word number.

6.2.3 $\sin(x)$ on $[0, \pi/4]$

reduction: If $|x|$ is small enough (the threshold chosen by Tang in [449] is $1/16$), then $\sin x$ is approximated by a polynomial. Otherwise, we find the "breakpoint" $c_{jk} = 2^{-j}(1 + k/8)$, with $j = 1, 2, 3, 4$ and $k = 0, 1, 2, \ldots, 7$, that is closest to x. Define $r = x - c_{jk}$. We have $|r| \leq 1/32$.

approximation: We approximate $\sin(r)$ and $\cos(r)$ by polynomials of the form $r + a_3 r^3 + a_5 r^5 + \cdots$ and $1 + a_2 r^2 + a_4 r^4 + \cdots$ respectively, for instance, using one of the polynomials given in Table 6.4 for the sine, and one of the polynomials given in Table 6.5 for the cosine.

reconstruction: We reconstruct $\sin(x)$ using:

$$\sin(x) = \sin(c_{jk})\cos(r) + \cos(c_{jk})\sin(r).$$

Table 6.4 *Approximations to* $\sin(r)$ *on* $[-1/32, 1/32]$ *with binary64 coefficients.*

Degree	Relative error	Polynomial
3	1.380×10^{-9}	r $-6004555168758405 \times 2^{-55} \cdot r^3$
5	7.300×10^{-15}	r $-6004799500178007 \times 2^{-55} \cdot r^3$ $+2401841613992359 \times 2^{-58} \cdot r^5$
7	2.366×10^{-20}	r $-1501199875790161 \times 2^{-53} \cdot r^3$ $+75059993762881 \times 2^{-53} \cdot r^5$ $-3659972533832221 \times 2^{-64} \cdot r^7$
9	5.392×10^{-22}	r $-6004799503160661 \times 2^{-55} \cdot r^3$ $+2401919801250219 \times 2^{-58} \cdot r^5$ $-1830033418144863 \times 2^{-63} \cdot r^7$ $+1603920471354595 \times 2^{-69} \cdot r^9$

Table 6.5 *Approximations to* $\cos(r)$ *on* $[-1/32, 1/32]$ *with binary64 coefficients.*

Degree	Relative error	Polynomial
4	5.111×10^{-14}	1 $-9007199244301055 \times 2^{-54} \cdot r^2$ $+3002262920973185 \times 2^{-56} \cdot r^4$
6	2.130×10^{-19}	1 $-4503599627370457 \times 2^{-53} \cdot r^2$ $+6004799499326321 \times 2^{-57} \cdot r^4$ $-3202452033253347 \times 2^{-61} \cdot r^6$
8	9.766×10^{-25}	1 $-1/2 \cdot r^2$ $+3002399751580325 \times 2^{-56} \cdot r^4$ $-6405119468272123 \times 2^{-62} \cdot r^6$ $+7319966033052175 \times 2^{-68} \cdot r^8$
10	3.381×10^{-26}	1 $-1/2 \cdot r^2$ $+6004799503160661 \times 2^{-57} \cdot r^4$ $-3202559734995187 \times 2^{-61} \cdot r^6$ $+7320132211411081 \times 2^{-68} \cdot r^8$ $-2539688046502537 \times 2^{-73} \cdot r^{10}$

6.3 Gal's Accurate Tables Method

This method is due to Gal [198] and was implemented for IBM/370-type machines [2]. A more recent implementation, especially suited for machines using the IEEE-754 floating-point arithmetic, was described by Gal and Bachelis [199] in 1991. It is very attractive when the accuracy used for the intermediate calculations is equal to the target accuracy.[2] It consists of tabulating the function

[2]We call *target accuracy* the accuracy of the number system used for representing the results.

being computed at *almost*-regularly spaced points that are "machine numbers" (i.e., that are exactly representable in the floating-point system being used), where the value of the function is *very close* to a machine number.[3] By doing this, we simulate a *larger accuracy*. Let us consider the following toy example.

Example 8 (computation of the exponential function). *Assume that we use a base*-10 *floating-point system with* 4*-digit significands, and that we want to evaluate the exponential function on the interval* $[1/2, 1]$. *A first solution is to store the five values* $e^{0.55}$, $e^{0.65}$, $e^{0.75}$, $e^{0.85}$, *and* $e^{0.95}$ *in a table, and to approximate, in the interval* $[i/10, (i+1)/10](i = 5, \ldots, 9)$, *the exponential of* x *by* $\exp((i+1/2)/10)$ *(which is stored) plus — or times — a polynomial function of* $y = x - \frac{i+1/2}{10}$. *The values stored in the table are:*

x	e^x	*Value Stored*	$\mid$ *Absolute error* $\mid$
0.55	$1.733253\cdots$	1.733	2.53×10^{-4}
0.65	$1.915540\cdots$	1.916	4.60×10^{-4}
0.75	$2.117000\cdots$	2.117	1.7×10^{-8}
0.85	$2.339646\cdots$	2.340	3.54×10^{-4}
0.95	$2.585709\cdots$	2.586	2.91×10^{-4}

The rounding error committed when storing the values is 4.6×10^{-4} *in the worst case, and has an average value equal to* 2.7×10^{-4}. *Now let us try to use Gal's method. We store the values of the exponential at points* X_i *that satisfy the following conditions.*

1. *They should be exactly representable in the number system being used (base* 10*,* 4*-digit significands);*
2. *they should be close to the values that were previously stored;*
3. e^{X_i} *should be very close to a number that is exactly representable in the number system being used.*

Such values can be found by an exhaustive or a random search. One can take the values:

X_i	e^{X_i}	*Value Stored*	$\mid$ *Error* $\mid$
0.5487	$1.73100125\cdots$	1.731	1.25×10^{-6}
0.6518	$1.91899190\cdots$	1.919	8.10×10^{-6}
0.7500	$2.11700001\cdots$	2.117	1.7×10^{-8}
0.8493	$2.33800967\cdots$	2.338	9.67×10^{-6}
0.9505	$2.58700283\cdots$	2.587	2.83×10^{-6}

Now, the rounding error becomes 9.67×10^{-6} *in the worst case, and has an average value equal to* 4.3×10^{-6}. *Thus this table is* 60 *times more accurate than the previous one for the average case, and* 47 *times for the worst case.*

Now let us study Gal and Bachelis' algorithm for computing sines and cosines in IEEE-754 binary64/double-precision floating-point arithmetic [199]. We assume that a range reduction has been

[3] We must notice that with the most common functions (exp, ln, sin, cos, arctan), there are no nontrivial machine numbers where the value of the function is *exactly* a machine number. This is a consequence of a theorem due to Lindemann, which shows that the exponential of a possibly complex algebraic nonzero number is not algebraic. This property is also used in Chapter 12.

performed (see Chapter 11), so that our problem is reduced to evaluating the sine or cosine of $u + du$, where u is a "machine number" of absolute value less than $\pi/4$, and du is a correction term, much smaller than u.

- To compute $\sin(u + du)$, if u is small enough (in Gal and Bachelis' method the bound is $83/512$), it suffices to use a polynomial approximation of the form

$$(((C_9 \times u2 + C_7) \times u2 + C_5) \times u2 + C3) \times u2 \times u + du + u,$$

  where $u2 = u^2$. Now, if u is larger, we must use "accurate tables." Gal and Bachelis use values $\sin(X_i)$ and $\cos(X_i)$, where the terms $X_i = i/256 + \epsilon_i$ (for $16 \leq i \leq 201$), are chosen so that $\sin(X_i)$ and $\cos(X_i)$ should contain at least 11 zeros after bit 53. After this, if i is the integer that is nearest to $256u$, and if[4] $z = (u - X_i) + du$, we use the well-known formula

$$\sin(X_i + z) = \sin(X_i) \times \cos(z) + \cos(X_i) \times \sin(z),$$

  where $\sin(z)$ and $\cos(z)$ are evaluated using polynomial approximations of low degrees (4 for $\cos(z)$ and 5 for $\sin(z)$).
- To compute $\cos(u + du)$, in a similar fashion, Gal and Bachelis use a polynomial approximation if u is small enough, and the formula

$$\cos(X_i + z) = \cos(X_i) \times \cos(z) - \sin(X_i) \times \sin(z)$$

  for larger values of u.

Using this method, Gal and Bachelis obtained last-bit accuracy in about 99.9 % of the tested cases. This shows that for software implementation, a careful programming of the accurate tables method is an interesting choice.[5]

Now let us concentrate on the cost of producing such accurate tables. We are looking for "machine numbers" X_i such that the value of one or more functions[6] at the point X_i is very close to a machine number. Let us assume that we use a p-bit-significand, radix-2, floating-point number system. We assume that if f is one of the usual elementary functions and x is a machine number, then the bits of $f(x)$ after position p can be viewed as if they were random sequences of 0s and 1s, with probability $1/2$ for 0 as well as for 1 (we make the same assumption in Chapter 12 for studying the possibility of always computing correctly rounded results). We also assume that when we need to tabulate several functions $f_1, f_2, \ldots, f_k$, the bits of $f_1(x), f_2(x), \ldots, f_k(x)$ after position p can be viewed as "independent."

Using our assumptions, one can easily see that the "probability" of having n zeros or n ones after position p in $f_1(x), f_2(x), \ldots, f_k(x)$ is 2^{-kn+k}. There are two consequences of this:

- if we want q such values X_i, and if we try to find them by means of an exhaustive or random search, we will have to try around $q \times 2^{kn-k}$ values. For instance, to find values X_i suitable for Gal and Bachelis' algorithm for sines and cosines (given previously), for which $k = 2$, $q = 185$, and $n = 11$, we have to try a number of values whose order of magnitude is 2×10^8;

[4] A consequence of Sterbenz Lemma—Theorem 1, page 15—is that $u - X_i$ is computed exactly.

[5] But it seems difficult to *always* get correctly rounded results without performing the intermediate calculations using or simulating an accuracy that is significantly larger than the target accuracy.

[6] We need one function for computing exponentials, two for computing sines or cosines.

- we want the values X_i to be "almost regularly spaced;" that is, each value X_i must be located in an interval of size, say, 2ϵ. Assuming that in this interval all floating-point numbers have the same exponent α, the number of machine numbers included in the interval is around $2^{p-\alpha}\epsilon$. Therefore, to make sure that there exist such values X_i, 2^{kn-k} must be small compared to $2^{p-\alpha}\epsilon$.

Luther [324] described a fast method for obtaining highly accurate tables by using Bresenham's algorithm. Stehlé and Zimmermann suggested significant improvements to the accurate tables method [434]. Using algorithms originally designed for finding worst cases of the table maker's dilemma (see Chapter 12 and references [306, 308, 311, 433]), they generate values X_i that are extremely good values, faster than when using Gal's original searching method.

6.4 Use of Pythagorean Triples

The following method was suggested by de Lassus Saint-Geniès, Defour, and Revy [141]. It is adapted to the calculation of trigonometric functions and could probably be extended to hyperbolic functions. Assume an input value x of absolute value less than $\pi/2$. That input value will in general be the result of a preliminary range reduction (see Chapter 11), and we assume that it is represented by a double-word number (see Section 2.2.2) $x_h + x_\ell$. De Lassus et al. assume that we have stored in table values $\sin(T_i)$ and $\cos(T_i)$ for which we have *exactly*

$$\begin{aligned} \sin(T_i) &= s_i/k, \\ \cos(T_i) &= c_i/k, \end{aligned} \tag{6.2}$$

where s_i, c_i, and k are integers (we even require s_i and c_i to fit into one word), and k is the same for all entries of the table. In practice the numbers T_i are irrational numbers, so that we only store approximations to them, and yet, the sines and cosines of the T_is are rational numbers with the same denominator.

A few most significant bits of x_h will be used to locate into the table a number T_i that is almost[7] nearest x_h, to obtain the corresponding values $s_i = k \cdot \sin(T_i)$ and $c_i = k \cdot \cos(T_i)$, and to compute a "correcting term" $\delta \approx T_i - x_h$, so that

$$x = x_h + x_\ell \approx T_i + (x_\ell - \delta).$$

The sine and cosine of x are then approximated by

$$s_i \cdot P_{\cos}(x_\ell - \delta) + c_i \cdot P_{\sin}(x_\ell - \delta)$$

and

$$c_i \cdot P_{\cos}(x_\ell - \delta) - s_i \cdot P_{\sin}(x_\ell - \delta),$$

where $P_{\cos}(t)$ and $P_{\sin}(t)$ are polynomial approximations to $\cos(t)/k$ and $\sin(t)/k$ respectively, valid for $|t|$ less than or equal to the maximum possible value of $|x_\ell - \delta|$. All the difficulty is to make that maximum possible value small, which implies finding values T_i that are well distributed in $[-\pi/2, +\pi/2]$. Lassus Saint-Geniès, Defour, and Revy suggest an algorithm for that based on the properties of Pythagorean triples (i.e., triples (a, b, c) of integers satisfying $a^2 + b^2 = c^2$).

[7] Requiring T_i to be *the* element of the table nearest x_h would require using all the bits of x_h.

6.5 Table Methods Requiring Specialized Hardware

The methods previously presented in this chapter may be implemented in software as well as in hardware. Now if we want to take advantage of a hardware implementation, we may try to find methods that are faster, but that require specialized hardware. Wong and Goto [475] suggested using a specialized hardware for implementing a table-based method. To evaluate the elementary functions in the IEEE-754 binary64/double-precision floating-point format (see Chapter 2), they use a *rectangular multiplier*, more precisely, a 16×56-bit multiplier[8] that truncates the final result to 56 bits. As they point out, such multipliers have already been implemented in floating-point processors: the Cyrix 83D87 coprocessor had a 17×69 bit multiply and add array [62]. A rectangular multiplier is faster than a full binary64 multiplier. According to Wong and Goto, the rectangular multipliers required by their algorithms operate in slightly more than half the time required to perform a full binary64 multiplication. Let us now examine some of Wong and Goto's algorithms.

6.5.1 Wong and Goto's Algorithm for Computing Logarithms

Assume we wish to compute the logarithm of a normal IEEE-754 binary64 floating-point number:

$$x = m \times 2^{exponent}.$$

We evaluate $\ln x$ as follows. First, we evaluate $\ln m$. Then we add $\ln m$ and $exponent \times \ln 2$ (the number $exponent \times \ln 2$ can be found in a table or computed using a rectangular multiplier). If $exponent = -1$, this final addition may lead to a *catastrophic cancelation*.[9] To avoid this, we assume $exponent \neq -1$: if $exponent = -1$, then m is one bit right-shifted, and $exponent$ is replaced by zero. As a consequence, we must assume[10]:

$$\frac{1}{2} \leq m < 2.$$

In the following,

$$[z]_{a-b}$$

denotes the number obtained by zeroing all the bits of z but the bits a to b. For instance, if $m = m_0.m_1m_2m_3m_4\ldots$, then

$$[m]_{1-3} = 0.m_1m_2m_3000\ldots.$$

[8] For most of their algorithms, a 12×56 bit rectangular multiplier suffices.

[9] A catastrophic cancelation is the total loss of accuracy that may occur when two numbers that are very close to each other are subtracted. It is worth noticing that there may be an error only in the case where at least one of the operands is inexact (i.e., it is the rounded result of a previous computation). Otherwise, the result of the subtraction is exact. As we have seen in Chapter 2 (Theorem 1), if x and y are floating-point numbers in the same format such that $x/2 \leq y \leq 2x$, and if the arithmetic operations are correctly rounded, then the subtraction $y - x$ is exactly performed.

[10] But we must keep in mind that $1/2 \leq m < 1$ if $exponent = 0$ only; otherwise, we could have a catastrophic cancelation for $exponent = 1$ and m close to $1/2$.

Let us focus on the computation of $\ln m$. The basic trick consists of finding a number K_1 such that $K_1 m \approx 1$, and such that the multiplication $K_1 \times m$ can be performed using a rectangular multiplier (i.e., K_1 must be representable with a few bits only, and close to $1/m$). This gives:

$$\ln(m) = \ln(K_1 m) - \ln(K_1),$$

where $\ln(K_1)$ is looked up in a table. Then we continue: we find a number K_2 such that $K_1 K_2 m$ is even closer to 1. After this our problem is reduced to evaluate the logarithm of $(K_1 K_2 m)$. We continue until our problem is reduced to the evaluation of a number that is so close to 1 that a simple low-order Taylor expansion suffices to get its logarithm. Now let us give the algorithm with more details (proofs can be found in [475]).

1. First, we obtain from tables the numbers

$$K_1 = \left[\frac{1}{[m]_{0-10} + 2^{-10}} \right]_{0-10}$$

and

$$[\ln(K_1)]_{1-56};$$

2. using a rectangular multiplier, we multiply m by K_1. The result is equal to $1-\alpha$, where $0 < \alpha < 2^{-8}$. Since we now want to find a value K_2 such that $K_1 K_2 m$ is very close to 1, $K_2 = 1 + \alpha$ would be convenient (this would give $|K_1 K_2 m - 1| = |(1 - \alpha^2) - 1| < 2^{-16}$). Unfortunately, multiplying $K_1 m$ by $1 + \alpha$ would require a full binary64 multiplier. To avoid this, we choose a slightly different value of K_2, representable with 10-bits. If α is equal to $0.00000000\alpha_9\alpha_{10}\cdots\alpha_{18}\alpha_{19}\cdots$, we define a 10-bit number $a = 0.00000000a_9a_{10}\cdots a_{18}$:

$$a_9a_{10}\cdots a_{18} = \begin{cases} \alpha_9\alpha_{10}\cdots\alpha_{17}\alpha_{18} + 1 & \text{if } \alpha_9 = 1 \\ \alpha_9\alpha_{10}\cdots\alpha_{17}0 & \text{otherwise,} \end{cases} \tag{6.3}$$

and we choose K_2 equal to $1.00000000a_9a_{10}\cdots a_{18}$. Then, from tables, we obtain $[\ln(K_2)]_{1-56}$.
3. Using a rectangular multiplier, we multiply (mK_1) by K_2. The result is equal to $1-\beta$, where (thanks to (6.3)) $0 \le \beta < 2^{-16}$. From tables, we obtain

$$[\ln(1.0000000000000000\beta_{17}\beta_{18}\beta_{19}\cdots\beta_{26})]_{1-56}.$$

Now, as in the previous step, we define a 10-bit number

$$b = 0.0000000000000000b_{17}b_{18}\cdots b_{26}$$

as follows

$$b_9b_{17}\cdots b_{26} = \begin{cases} \beta_{17}\beta_{18}\beta_{19}\cdots\beta_{25}\beta_{26} + 1 & \text{if } \beta_{17} = 1 \\ \beta_{17}\beta_{18}\beta_{19}\cdots\beta_{25}0 & \text{otherwise,} \end{cases} \tag{6.4}$$

and we define K_3 as $1.0000000000000000b_{17}b_{18}b_{19}\cdots b_{26}$. From tables, we obtain $[\ln K_3]_{1-56}$.

4. Using a rectangular multiplier, we multiply (mK_1K_2) by K_3. The result is equal to $1-\gamma$, where $0 \le \gamma < 2^{-24}$. Now $1-\gamma$ is so close to 1 that the degree-3 Taylor approximation

$$\ln(1-\gamma) \approx -\gamma - \frac{\gamma^2}{2} - \frac{\gamma^3}{3}$$

suffices to get its logarithm with good accuracy.

5. Using a *full multiplication*, we compute

$$\left[\frac{\gamma^2}{2}\right]_{1-56}$$

and from tables, we obtain

$$\left[\frac{[\gamma]^3_{25-33}}{3}\right]_{1-56}.$$

6. We then compute the final result:

$$\ln(x) \approx exponent \times \ln(2) - \ln K_1 - \ln K_2 - \ln K_3 - \gamma - \left[\frac{\gamma^2}{2}\right]_{1-56} - \left[\frac{[\gamma]^3_{25-33}}{3}\right]_{1-56}.$$

In [475] Wong and Goto give an error analysis of this algorithm, and claim that the error is within 0.5 ulps (see Chapter 2 for an explanation of what a ulp is). This is slightly misleading. They actually compute a 56-bit number that is within 0.5 ulps *of the "target format"* (i.e., the 53-significand bit IEEE binary64 format). When rounding the 56-bit result to the target format, an extra rounding error is committed; hence the final 53-bit result is within 1 ulp of the exact result. Figure 6.1 illustrates this. Getting correctly rounded results would require an intermediate computation with many more than 56 bits (see Chapter 12 for a discussion of this topic).

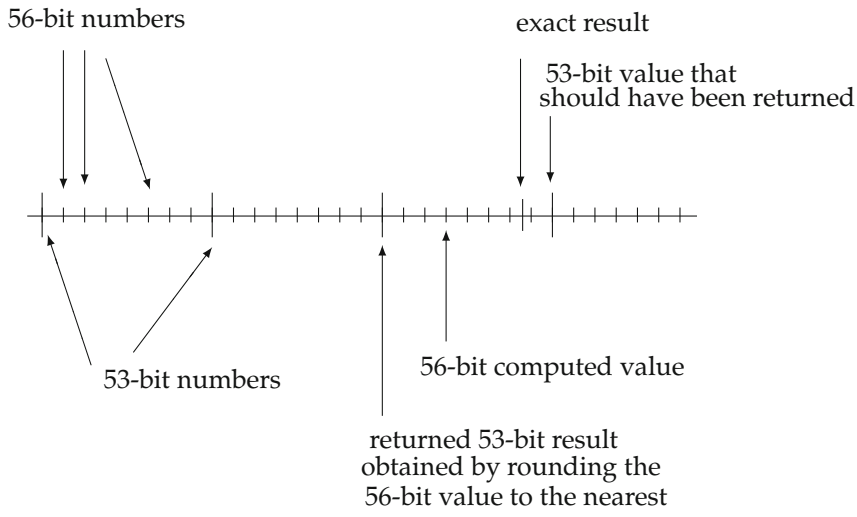


Figure 6.1 *An incorrectly rounded result deduced from a 56-bit value that is within 0.5 ULPs from the exact result. We assume that rounding to the nearest was desired.*

The algorithm we have given here computes natural logarithms. Slight (and rather straightforward) modifications would give algorithms for base-2 or base-10 logarithms. Adaptations to smaller (single-precision) or larger (extended or quad-precision) formats can also be derived.

6.5.2 Wong and Goto's Algorithm for Computing Exponentials

Assume that we want to compute e^x, where x is a normal binary64/double-precision floating-point number:

$$x = s \times m \times 2^{exponent}.$$

The largest representable finite number in the IEEE-754 binary64 format is

$$\left(1 - 2^{-53}\right) \times 2^{2^{10}} \approx 1.7976931348623157 \times 10^{308},$$

the logarithm of which is $A = 709.7827128933839967\cdots$. Therefore, if x is larger than A, when we evaluate $\exp(x)$, we must return[11] $+\infty$.

Therefore we can assume that x can be rewritten as the sum of a 10-bit integer and a 56-bit fixed-point fractional number:

$$x \approx \hat{x} = s \times (h + 0.x_1x_2x_3\cdots x_{56}), \, 1 \le h < 2^{10}, s = \pm 1.$$

If $|x| \ge 1/8$, this is done without any loss of accuracy, and if $|x| < 1/8$, we have

$$\left|e^x - e^{\hat{x}}\right| \le 2^{-57}e^{1/8} \le 1.14 \times 2^{-57},$$

assuming that x is rounded to the nearest to get $\hat{x}$. Therefore we can replace x by $\hat{x}$.

The basic idea behind the algorithm consists of rewriting the number $0.x_1x_2x_3\cdots x_{56}$ as a sum

$$\alpha_1 + \alpha_2 + \cdots + \alpha_n,$$

where n is small, and where the α_is are representable using a few bits only, so that we can look e^{α_i} up in a table of reasonable size. Then

$$e^x = e^h \times f_1 \times f_2 \times \cdots \times f_n, \tag{6.5}$$

where $f_i = e^{\alpha_i}$.

Unfortunately, this cannot be done without modifications since (6.5) would require full binary64 multiplications. We have to get values f_i that are representable with a few bits only. To do this we proceed as follows. First, α_1 is the number constituted by the first 9 bits of $0.x_1x_2x_3\cdots x_{56}$. Second, we look up in a table the value:

$$f_1 = \left[e^{\alpha_1}\right]_{(-1)-9}.$$

[11] Unless we wish to evaluate the exponential function in round towards 0 or round towards $-\infty$ mode: in such a case, we must return the largest finite representable number, that is, $\left(1 - 2^{-53}\right) \times 2^{2^{10}}$, unless x is equal to $+\infty$.

Using a rectangular multiplier, we can compute $K_1 = e^h \times f_1$. Since f_1 is an approximation to e^{α_1} only, e^x is not equal to $K_1 \times e^{0.000000000x_{10}x_{11}\cdots x_{56}}$, therefore we have to perform a slight correction by looking up in a table $\ln f_1 \approx \alpha_1$, and computing

$$r_1 = 0.x_1x_2x_3\cdots x_{56} - \ln f_1.$$

After this, $e^x = K_1 \times e^{r_1}$, and we continue by choosing α_2 equal to the 9 most significant bits of r_1. Let us now give the algorithm with more details.

1. Rewrite x in a 56-fractional-bit, 10-integer-bit fixed-point representation:

$$x = s \times (h + 0.x_1x_2x_3\cdots x_{56}), 1 \le h < 2^{10}.$$

   If x is too large to do that, return $+\infty$ (if $s = +1$) or 0 (if $s = -1$), or the largest representable number, or the smallest nonzero positive number, depending on the rounding mode.
2. Look the following values up from tables.

   - $f_0 = e^h$ if $s = +1$, or e^{-h} if $s = -1$,
   - $f_1 = [e^{\alpha_1}]_{0-9}$, where $\alpha_1 = 0.x_1x_2\cdots x_9$,
   - $lf_1 = [\ln f_1]_{1-56}$.

3. Compute $r_1 = 0.x_1x_2\cdots x_{56} - lf_1$. One can show that $r_1 < 2^{-8}$. Look the following values up from tables.

   - $f_2 = [e^{\alpha_2}]_{0-17}$, where $\alpha_2 = 0.00000000r_{1,9}r_{1,10}r_{1,11}\cdots r_{1,17}$,
   - $lf_2 = [\ln f_2]_{1-56}$.

4. Compute $r_2 = r_1 - lf_2$. One can show that $r_2 < 2^{-16}$. Then get the final result:

$$e^x \approx f_0 \times f_1 \times f_2 \times \left[1 + r_2 + \frac{[r_2]_{17-24}}{2}\left([r_2]_{25-32} + [r_2]_{33-40}\right) + \frac{[r_2]^2_{17-24}}{2} + \frac{[r_2]^2_{25-32}}{2} + \frac{[r_2]^3_{17-24}}{6}\right].$$

6.5.3 Ercegovac et al.'s Algorithm

Let f be a function whose first Taylor coefficients C_0, C_1, C_2, and C_3 are either powers of two, or very simple rational numbers (so that multiplication by C_i is straightforward). For computing f, with around k bits of accuracy, Ercegovac, Lang, Muller, and Tisserand [170] suggest to first perform a range reduction so that the input argument A becomes less than 2^{-k}. For computation of reciprocals, square roots and square root reciprocals, this is done by computing

$$A = Y \times \hat{R} - 1,$$

where Y is the initial input argument (assumed between 1 and 2) and $\hat{R}$ is a $(k+1)$-bit approximation to $1/Y$, obtained through table lookup in a table addressed by the first k bits of Y.

After that, for computing $f(A)$, where

$$A = A_2 z^2 + A_3 z^3 + A_4 z^4 + \cdots, \text{ with } z = 2^{-k},$$

and the A_i are k-bit numbers, and using the Taylor expansion of f

$$f(A) = C_0 + C_1 A + C_2 A^2 + C_3 A^3 + \cdots, \tag{6.6}$$

they use the following formula:

$$f(A) \approx C_0 A + C_1 A + C_2 A_2^2 z^4 + 2C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6. \tag{6.7}$$

That formula is obtained by expanding the series (6.6) and dropping the terms of the form $W z^j$ that are less than or equal to 2^{-4k}.

For instance, for square root, we get

$$\sqrt{1+A} \approx 1 + \frac{A}{2} - \frac{1}{8} A_2^2 z^4 - \frac{1}{4} A_2 A_3 z^5 + \frac{1}{16} A_2^3 z^6,$$

while for reciprocals, we get

$$\frac{1}{1+A} \approx (1 - A) + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6.$$

In practice, when computing (6.7), another approximation is made: to compute A_2^3, once A_2^2 is obtained, we take the k most significant bits of A_2^2 only, and multiply them by A_2. Figure 6.2 presents the "evaluation module" that computes $f(A)$.

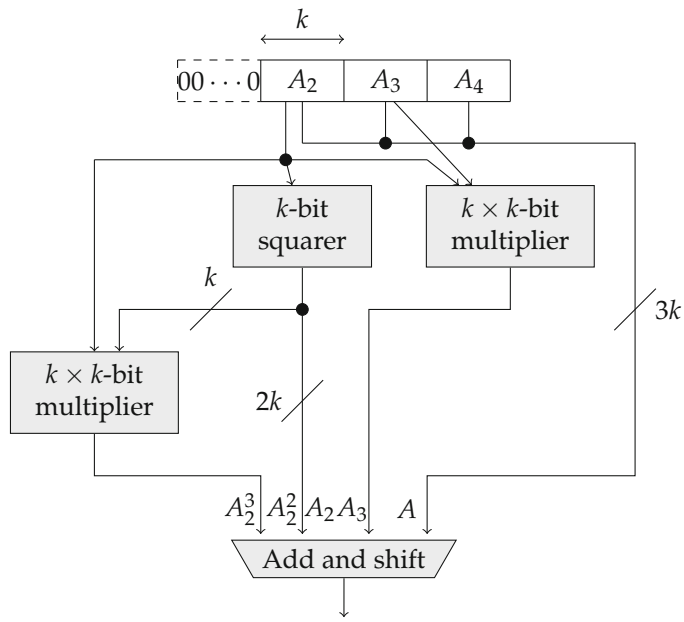


Figure 6.2 *The computation of $f(A)$ using Ercegovac et al.'s algorithm.*

6.5.4 Bipartite and Multipartite Methods

In [437], Sunderland et al. needed to approximate the sine of a 12-bit number x less than $\pi/2$ using tables. They decided to split the binary fixed-point representation of x into three 4-bit words, and to approximate the sine of $x = A + B + C$, where $A < \pi/2$, $B < 2^{-3}$ and $C < 2^{-7}$, using

$$\sin(A + B + C) \approx \sin(A + B) + \cos(A)\sin(C). \tag{6.8}$$

By doing that, instead of one table with 12 address bits (i.e., with 2^{12} elements), one needed two tables (one for $\sin(A + B)$ and one for $\cos(A)\sin(C)$), each of them with 8 address bits only. To my knowledge, this was the first use of what is now called the *bipartite method*.

That method was rediscovered (and named bipartite) by DasSarma and Matula [409]. Their aim was to quickly generate seed values for computing reciprocals using the Newton–Raphson iteration.

Assume we want to evaluate function f and that the input and output values are represented on a p-bit fixed-point format and belong to $[\alpha, 1)$, where $\alpha > 0$ is some real number. Also, assume $p = 3k$. A straightforward tabulation of function f would require a table with p address bits. The size of that table would be $p \times 2^p$ bits. The first idea behind the bipartite method consists of splitting the input p-bit value x into three k-bit subwords x_0, x_1, and x_2. We have

$$x = x_0 + 2^{-k}x_1 + 2^{-2k}x_2$$

where x_i is a multiple of 2^{-k} and $0 \le x_i < 1$. The order-1 Taylor expansion of f near $x_0 + 2^{-k}x_1$ gives

$$f(x) = f(x_0 + 2^{-k}x_1) \; + \; 2^{-2k}x_2.f'(x_0 + 2^{-k}x_1) \; + \; \epsilon_1$$

where $|\epsilon_1| \le 2^{-4k-1} \max_{[\alpha,1]} |f''|$. Now, we can replace $f'(x_0+2^{-k}x_1)$ by its order-0 Taylor expansion near x_0:

$$f'(x_0 + 2^{-k}x_1) = f'(x_0) + \epsilon_2$$

where $|\epsilon_2| \le 2^{-k} \max_{[\alpha,1]} |f''|$. This gives the *bipartite formula*:

$$\begin{array}{ll} f(x) & = A(x_0, x_1) + B(x_0, x_2) + \epsilon \\ \text{where} & \\ A(x_0, x_1) & = f(x_0 + 2^{-k}x_1) \\ B(x_0, x_2) & = 2^{-2k}x_2.f'(x_0) \\ \epsilon & \le \left(2^{-4k-1} + 2^{-3k}\right) \max_{[\alpha,1]} |f''|. \end{array} \tag{6.9}$$

If the second derivative of f has the same order of magnitude as f itself, then the error ϵ of this approximation is of the order of the representation error of this number system (namely around 2^{-3k}). The second idea behind the bipartite method consists of tabulating A and B instead of tabulating f. Since A and B are functions of $2k = 2p/3$ bits only, if A is rounded to the nearest p-bit number, if $2^{2k}B$ is rounded to the nearest $k = p/3$-bit number, and if the values of A and B are less than 1, then the total table size is

$$\left(\frac{4p}{3}\right) 2^{2p/3} \text{ bits}$$

which is a very significant improvement. Here, we implicitly assume that f, f', and f'' have orders of magnitude that are close together. In other cases, the method must be modified. For $p = 15$, the table size is 60 kbytes with a straightforward tabulation, and 2.5 kbytes with the bipartite method. The error of the approximation will be ϵ plus the error due to the rounding of functions A and B. It will then be bounded by

$$\left(2^{-4k-1} + 2^{-3k}\right) \max_{[\alpha,1]} |f''| + 2^{-3k} \left(\max_{[\alpha,1]} |f| + \max_{[\alpha,1]} |f'| \right).$$

For instance, for getting approximations to $1/x$ (which was Das Sarma and Matula's goal), we choose $f(x) = 1/x$ and $\alpha = 1/2$. Hence,

$$\begin{cases} A(x_0, x_1) = 1/(x_0 + 2^{-k} x_1) \\ B(x_0, x_2) = -2^{-2k} x_2 / x_0^2 \\ \epsilon \qquad\quad \le 16 \left(2^{-4k-1} + 2^{-3k}\right). \end{cases}$$

To get better approximations, one can store A and B with some additional guard bits. Also, p can be chosen slightly larger than the actual word size. A compromise must be found between table size and accuracy. Of course, that compromise depends much on the function being tabulated.

Schulte and Stine [414, 415] gave the general formula (6.9). They also suggested several improvements. The major one was to perform the first Taylor expansion near[12] $x_0 + 2^{-k} x_1 + 2^{-2k-1}$ instead of $x_0 + 2^{-k} x_1$, and the second one near $x_0 + 2^{-k-1}$ instead of x_0:

- this makes the maximum possible absolute value of the remaining term (namely, $2^{-2k} x_2 - 2^{-2k-1}$) smaller, so that the error terms ϵ_1 and ϵ_2 become smaller;
- B becomes a symmetric function of $2^{-2k} x_2 - 2^{-2k-1}$. Hence we can store its values for the positive entries only. This allows us to halve the size of the corresponding table.

The bipartite method can be viewed as a piecewise linear approximation for which the slopes of the approximating straight lines are constants in intervals sharing the same value of x_0. This is illustrated by Figure 6.3. Let us now give an example.

Example 9 *(*Cosine function on $[0, \pi/4]$.*) Assume we wish to provide the cosine of* 16*-bit numbers between* 0 *and* $\pi/4$*. We use the bipartite method, with the improvements suggested by Schulte and Stine. Since* 16 *is not a multiple of* 3*, we choose* $k = 6$ *and we split* x *into subwords of different sizes:* x_0 *and* x_1 *are* 6*-bit numbers, and* x_2 *is a* 4*-bit number. The functions* A *and* B *that will be tabulated are*

$$\begin{cases} A(x_0, x_1) = \cos(x_0 + 2^{-k} x_1 + 2^{-2k-1}) \\ B(x_0, x_2) = -(2^{-2k} x_2 - 2^{-2k-1}) \sin(x_0 + 2^{-k-1}). \end{cases}$$

If A *and* B *were stored exactly (which, of course, is impossible), the approximation error* ϵ *would be less than or equal to* $2^{-3k-2} + 2^{-4k-3} = 2^{-20} + 2^{-27}$*. Assume the tables for functions* A *and* B *store values correctly rounded up to the bit of weight* 2^{-20}*. The total error, including the rounding of the values stored in the tables, becomes less than* $2^{-19} + 2^{-27}$*. Since each word of* A *is stored with* 20 *bits of precision and each word of* B *is stored with* 8 *bits of precision (including the sign bit), the size of the tables is*

$$20 \times 2^{12} + 8 \times 2^{10} \text{ bits } = 11 \text{ kbytes}.$$

[12] That is, near the middle of the interval constituted by the numbers whose first bits are those of x_0 and x_1, instead of near the smallest value of that interval.

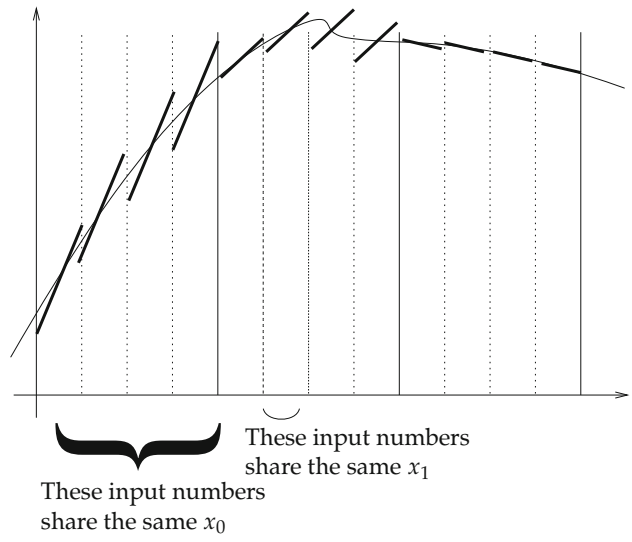


Figure 6.3 *The bipartite method is a piecewise linear approximation for which the slopes of the approximating straight lines are constants in intervals sharing the same value of* x_0.

Obtaining a similar accuracy with a direct tabulation would require a table with

$$18 \times 2^{16}\ bits\ = 144\ kbytes\ .$$

Schulte and Stine [416, 435] and Muller [355] independently generalized the bipartite table method to various multipartite table methods. With these methods, the results are obtained by summing values looked up in parallel in three tables or more. De Dinechin and Tisserand [138, 139] improved these multipartite methods and showed that optimal (in terms of total table size) multipartite table methods can be designed at a reasonable cost. A description of their method, as well as a Java program that builds the tables can be obtained at

http://www.ens-lyon.fr/LIP/Arenaire/Ware/Multipartite/.

The AMD-K7 floating-point unit used bipartite ROM tables for the reciprocal and square root initial approximations [366].

Matula and Panu [355] recently suggested to "prescale" the input value before using the bipartite algorithm to obtain a single-precision ulp accurate reciprocal. They applied this idea to division algorithms: generalization to elementary functions is still to be investigated.

6.6 (M, p, k)-Friendly Points: A Method Dedicated to Trigonometric Functions

The following method was introduced by Brisebarre, Ercegovac, and Muller [65], and later on improved and implemented by the same authors and Wang [472]. The underlying idea is the following. We first define "friendly points and angles" as follows.

Definition 2 *A pair of integers* (a, b) *is an* (M, p, k)*-friendly point if:*

1. $0 \leq a \leq M$ *and* $0 \leq b \leq M$*;*
2. *the number* $z = 1/\sqrt{a^2 + b^2}$ *can be written*

$$2^e \cdot 1.z_1 z_2 z_3 \cdots z_p z_{p+1} z_{p+2} \cdots = \sum z_i 2^{e-i},$$

where e *is an integer,* $z_i \in \{-1, 0, 1\}$*,* $z_1 \neq -1$*, and the number of terms* z_i *such that* $1 \leq i \leq p$ *and* $z_i \neq 0$ *is less than or equal to* k*.*

Definition 3 *The number* α*,* $0 \leq \alpha \leq \pi/2$ *is an* (M, p, k)*-friendly angle if either* $\alpha = 0$ *or* $\alpha = \arctan(b/a)$*, where* (a, b) *is an* (M, p, k)*-friendly point.*

Given an input angle x and parameters p, k and M, with (M, p, k)-friendly points precomputed and stored in a table, the algorithm for computing $\sin(x)$ and $\cos(x)$ consists of:

- looking up in a table, addressed by a few leading bits of x, an (M, p, k)-friendly angle $\hat{x}$ and the associated values a, b, and z (in canonical form—see Section 2.3.3);
- computing, using the bipartite method, $\sin(\theta)$ and $\cos(\theta)$, where $\theta = x - \hat{x}$;
- computing $C = a\cos(\theta) - b\sin(\theta)$ and $S = b\cos(\theta) + a\sin(\theta)$ using a very few additions/subtractions—since a and b are less than M, multiplying by a and b requires at most $\frac{1}{2}\lceil \log_2 M \rceil$ additions/subtractions that can be performed without carry propagation using redundant (e.g., carry-save) arithmetic;
- finally, multiplying C and S by z by adding a very few (at most k) multiples of C (S), using redundant (carry-free) adders followed by a carry-propagate adder (which may be omitted if the results can be used in redundant form).

What makes the method working is that, in practice, θ can be quite small for not-too-large values of the parameters M and k. For instance, for $M = 255$, $p = 24$, and $k = 5$, we can have $|\theta| < 2^{-6.353}$ [65].

Multiple-Precision Evaluation of Functions 7

7.1 Introduction

Multiple-precision arithmetic is a useful tool in many domains of contemporary science. Some numerical applications are known to sometimes require significantly more precision than provided by the usual binary32/single-precision, binary-64/double precision, and Intel extended-precision formats [22]. Some computations in Mathematical physics [19, 20, 31] or in "experimental mathematics" [19, 49] need to be performed with hundreds or thousands of bits of precision. For instance, multiple-precision calculations allowed Borwein, Plouffe, and Bailey to find an algorithm for computing individual[1] hexadecimal digits of π. The study of dynamical system also sometimes requires large precision [260]. Multiple-precision arithmetic is also an invaluable tool for tuning algorithms and approximations that will be used, later on, in basic precision arithmetic. For instance, when we design algorithms for computing functions in the binary32, binary64, extended, or binary128 floating-point formats, the various constants used in these algorithms (e.g., minimax polynomial or rational coefficients) need to be computed using a precision that is significantly higher than the target precision. For that purpose, the Gappa and Sollya tools presented in Chapters 2 and 3 use the GNU-MPFR multiple-precision package. Also, the availability of an efficient and reliable multiple-precision library is of great help for testing floating-point software [60]. It may be useful to mix multiple-precision arithmetic with interval arithmetic, where the intervals may become too large if we use standard (i.e., fixed-precision) interval arithmetic [349, 395]. Various examples of applications where multiple-precision is interesting can be found in [17, 19, 48].

A full account of multiple-precision calculation algorithms for the elementary functions is beyond the purpose of this book (it would probably require another full book). And yet, the reader may be interested in knowing, at least roughly, the basic principles of the algorithms used by the existing multiple-precision packages. Additional information can be found in [22, 480], and in the excellent book by Brent and Zimmermann [61].

In this domain, the pioneering work was done by Brent [56, 57] and Salamin [408]. Brent designed an arithmetic package named MP [58, 59]. It was the first widely diffused package that was portable and contained routines for all common elementary and special functions. Bailey designed three packages,

[1] That is, there is no need to compute any of the previous digits. For instance [49], the 2.5×10^{14}th hexadecimal digit of π is an E.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_7

MPFUN [15], ARPREC [22] and, more recently, MPFUN2015 [18].[2] The PARI package[3], originally designed by Cohen and his research group, was especially designed for number theory and multiple-precision. The GNU-MPFR[4] library [194] designed by Zimmermann and his research group is a C library for multiple-precision floating-point computations with correct rounding. It is based on Granlund's GMP [210] multiple-precision library. MPFI, written by Revol and Rouiller, is an extension of MPFR that implements multiple-precision interval arithmetic [395]. Sage[5] [30] is a free, open-source, computer algebra system that does multiple-precision calculations using GMP and GNU-MPFR.

In general, when the required precision is not huge (say, when we only want up to a few hundreds or thousands of bits), Taylor series of the elementary functions are used. For very large precision calculations, quadratically convergent methods that are based on the arithmetic–geometric mean (see Section 7.5) are sometimes used.

The number of digits used for representing data in a multiple-precision computation depends much on the problem being dealt with. Many numerical applications only require, for critical parts of calculations, a precision that is slightly larger than the available (binary64 or binary128) floating-point precisions (i.e., a few hundreds of bits only). On the other hand, experiments in number theory may require the manipulation of numbers represented by millions of bits.

7.2 Just a Few Words on Multiple-Precision Multiplication

It may seem strange to discuss multiplication algorithms in a book devoted to transcendental functions. Multiplication is not what we usually call an elementary function[6], but the performance of a multiple-precision system depends much on the performance of the multiplication programs: all other operations (including division, as we will see in Section 7.3) and functions use multiplications. For very low precisions, the grammar school method is used. When the desired precision is around a few hundreds of bits, Karatsuba's method (or one of its variants) is preferable. For very high precisions (thousands of bits), methods based on the Fast Fourier Transform (FFT) are used.

7.2.1 Karatsuba's Method

Assume we want to multiply two n-bit integers[7]

$$A = a_{n-1}a_{n-2}a_{n-3}\cdots a_0$$

and

$$B = b_{n-1}b_{n-2}b_{n-3}\cdots b_0,$$

[2] MPFUN2015 is available at http://www.davidhbailey.com/dhbsoftware/

[3] See http://pari.math.u-bordeaux.fr/.

[4] Available at http://www.mpfr.org.

[5] See http://www.sagemath.org

[6] In fact, it *is* an elementary function. The functions we deal with in this book should be called *elementary transcendental functions*.

[7] I give this presentation assuming radix 2 is used. Generalization to other radices is straightforward.

and assume that n is even. We want to compute AB using n/2-bit × n/2-bit multiplications. Define the following n/2-bit numbers, obtained by splitting the binary representations of A and B

$$\begin{cases} A^{(1)} = a_{n-1}a_{n-2}a_{n-3}\cdots a_{n/2} \\ A^{(0)} = a_{n/2-1}a_{n/2-2}a_{n/2-3}\cdots a_0 \\ B^{(1)} = b_{n-1}b_{n-2}b_{n-3}\cdots b_{n/2} \\ B^{(0)} = b_{n/2-1}b_{n/2-2}b_{n/2-3}\cdots b_0. \end{cases}$$

A is obviously equal to $2^{n/2}A^{(1)} + A^{(0)}$, and B is equal to $2^{n/2}B^{(1)} + B^{(0)}$. Therefore,

$$AB = 2^n A^{(1)}B^{(1)} + 2^{n/2}(A^{(1)}B^{(0)} + A^{(0)}B^{(1)}) + A^{(0)}B^{(0)}. \tag{7.1}$$

Hence, we can perform one $n \times n$-bit multiplication using four $(n/2) \times (n/2)$-bit multiplications. If we do that recursively (by decomposing the $(n/2) \times (n/2)$-bit multiplications into $(n/4) \times (n/4)$-bit multiplications and so on), we end-up with an algorithm that multiplies two n-bit numbers in a time proportional to n^2, which is not better than the usual pencil and paper algorithm. And yet, we can bring significant improvements.

In 1962, Karatsuba and Ofman [268] noticed that[8]

$$\begin{aligned} AB = {} & 2^n A^{(1)}B^{(1)} \\ & + 2^{n/2}\left((A^{(1)} - A^{(0)})(B^{(0)} - B^{(1)}) + A^{(0)}B^{(0)} + A^{(1)}B^{(1)}\right) \\ & + A^{(0)}B^{(0)}. \end{aligned} \tag{7.2}$$

Using (7.2), we can perform one $n \times n$-bit multiplication using *three* $(n/2) \times (n/2)$-bit multiplications only: we compute the products $A^{(1)}B^{(1)}$, $A^{(0)}B^{(0)}$, and $(A^{(1)} - A^{(0)})(B^{(0)} - B^{(1)})$. When this decomposition is applied recursively, this leads to an algorithm that multiplies two n-bit numbers in a time proportional to

$$n^{\frac{\ln(3)}{\ln(2)}} \approx n^{1.585}.$$

Montgomery [347] presented several Karatsuba-like formulae.

7.2.2 The Toom-Cook Family of Multiplication Algorithms

Karatsuba's method can be generalized. Similar (and asymptotically more efficient) decompositions of a product can be found in the literature [107, 275, 457, 483]. Let us present here the Toom-Cook family of algorithms. These algorithms are based on two ideas:

- transforming our initial problem of multiplying two integers into the problem of multiplying two adequately chosen polynomials;
- multiplying the two polynomials through a combination of polynomial evaluation and interpolation at adequately chosen points.

[8] This is not exactly Katatsuba and Ofman's presentation. I give here Knuth's version of the algorithm [275], which is slightly better.

Assume we wish to multiply two n-bit integers $a = a_{n-1}a_{n-2}\cdots a_0$ and $b = b_{n-1}b_{n-2}\cdots b_0$. We will first split the binary representations of a and b into m ν-bit parts:[9]

$$\underbrace{a_{n-1}a_{n-2}\cdots a_{n-\nu}}_{A_{m-1}}\underbrace{a_{n-\nu-1}a_{n-\nu-2}\cdots a_{n-2\nu}}_{A_{m-2}}\ \cdots\ \underbrace{a_{\nu-1}a_{\nu-2}\cdots a_0}_{A_0}$$

and

$$\underbrace{b_{n-1}b_{n-2}\cdots b_{n-\nu}}_{B_{m-1}}\underbrace{b_{n-\nu-1}b_{n-\nu-2}\cdots b_{n-2\nu}}_{B_{m-2}}\ \cdots\ \underbrace{b_{\nu-1}b_{\nu-2}\cdots b_0}_{B_0}.$$

The digit chains $A_{m-1}\cdots A_0$ and $B_{m-1}\cdots B_0$ are radix-2^ν representations of a and b. To these two digit chains we will associate two polynomials

$$A(X) = A_{m-1}X^{m-1} + A_{m-2}X^{m-2} + \cdots + A_0,$$

and

$$B(X) = B_{m-1}X^{m-1} + B_{m-2}X^{m-2} + \cdots + B_0.$$

We have $a = A(2^\nu)$ and $b = B(2^\nu)$. We would like to compute the coefficients of the polynomial $C(X) = A(X) \cdot B(X)$. This would almost immediately give the value of the product $c = ab$ since $c = C(2^\nu)$ and since the multiplications involved in the evaluation of a polynomial at point 2^ν are mere shifts.

We will first evaluate the polynomials A and B at $2m-1$ points $x_1, x_2, x_3, \ldots, x_{2m-1}$, chosen such that

- A and B are very easily evaluated at these points;
- the values $A(x_i)$ and $B(x_i)$ are very small in front of a and b;

and we will compute the value of C at these points. This will require the computation of $2m-1$ products, but these products will be multiplications of terms much smaller than a and b, so that computing these $2m-1$ products will be significantly faster than directly computing ab—and we can use the same strategy for computing these products recursively. In practice, the values x_i are very small integers (typically, $-2, -1, 0, 1, 2$). A convenient choice is also ∞ (which is just a way of saying that we take the coefficient of highest degree of the polynomial: we define $A(\infty)$ as A_{m-1}). Other possible choices could be the reciprocals of very small powers of 2, or $\pm i$, $\pm 2i$, etc. Notice that Karatsuba's method corresponds to the case $m = 2$ with the points $0, -1$, and ∞.

Since the polynomial C is of degree $2m-2$, knowing its value at $2m-1$ points suffices to find its coefficients (below, we are going to see how this is done). Let us consider an example. For some sizes of the input operands, the GMP software uses $m = 3$, with the following values[10] x_i: 0, 1, -1, 2, and ∞. We have

[9] To simplify we assume $n = m\nu$ exactly. In practice, if needed, we add a few zero bits at the left of the binary representations of a and b, and we choose $\nu = \lceil n/m \rceil$.

[10] See https://gmplib.org/manual/Toom-3_002dWay-Multiplication.html

$$\begin{aligned} A(0) &= A_0, \\ A(1) &= A_2 + A_1 + A_0, \\ A(-1) &= A_2 - A_1 + A_0, \\ A(2) &= 4A_2 + 2A_1 + A_0, \\ A(\infty) &= A_2. \end{aligned} \tag{7.3}$$

and a similar equation for the values $B(x_i)$. Hence, computing the values $A(x_i)$ and $B(x_i)$ requires a few additions and shifts only. Also, the largest absolute value that the terms $A(x_i)$ and $B(x_i)$ can take is $(4 + 2 + 1) \cdot (2^\nu - 1)$: hence $A(x_i)$ and $B(x_i)$ fit in at most $\nu + 3$ bits, which means that the computation of the terms $C(x_i) = A(x_i) \cdot B(x_i)$ is done through multiplication of $\lceil n/3 \rceil + 3$-bit numbers. We finally need to retrieve the coefficients of the polynomial C from the values $C(x_i)$. That polynomial is of degree 4, and if we denote C_i its degree-i coefficient, we have

$$\begin{aligned} C(0) &= C_0, \\ C(1) &= C_4 + C_3 + C_2 + C_1 + C_0, \\ C(-1) &= C_4 - C_3 + C_2 - C_1 + C_0, \\ C(2) &= 16C_4 + 8C_3 + 4C_2 + 2C_1 + C_0, \\ C(\infty) &= C_4. \end{aligned} \tag{7.4}$$

Equation (7.4) is a linear system of Matrix

$$\mathcal{M} = \begin{pmatrix} 0 & 0\,0 & 0\,1 \\ 1 & 1\,1 & 1\,1 \\ 1 & -1\,1 & -1\,1 \\ 16 & 8\,4 & 2\,1 \\ 1 & 0\,0 & 0\,0 \end{pmatrix}.$$

Matrix $\mathcal{M}$ is invertible (it is a special case of a Vandermonde matrix [327]), and its inverse is (fortunately!) very simple

$$\mathcal{M}^{-1} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1/2 & -1/2 & -1/6 & 1/6 & -2 \\ -1 & 1/2 & 1/2 & 0 & -1 \\ -1/2 & 1 & -1/3 & -1/6 & 2 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix},$$

so that

$$\begin{aligned} C_4 &= C(\infty), \\ C_3 &= \tfrac{1}{2}C(0) - \tfrac{1}{2}C(1) - \tfrac{1}{6}C(-1) + \tfrac{1}{6}C(2) - 2C(\infty), \\ C_2 &= -C(0) + \tfrac{1}{2}C(1) + \tfrac{1}{2}C(-1) - C(\infty), \\ C_1 &= -\tfrac{1}{2}C(0) + C(1) - \tfrac{1}{3}C(-1) - \tfrac{1}{6}C(2) + 2C(\infty), \\ C_0 &= C(0). \end{aligned} \tag{7.5}$$

In (7.5), the multiplications by 2 or $1/2$ are simple shifts. The division by 3 is very easily done too (and since we know that the final value C_i is an integer, we do not need to compute it with many fractional bits). The division by 6 is a one bit shift followed by a division by 3.

To sum-up, the multiplication of two n-bit numbers reduces to five multiplications (the products $A(x_i) \cdot B(x_i)$) of $\lceil n/3 \rceil + 3$-bit numbers. If we apply the same strategy recursively to these five products, we end up with a time proportional to

$$n^{\frac{\log(5)}{\log(3)}} \approx n^{1.465}.$$

In the general case (i.e., an arbitrary value of m), we obtain an algorithm with a delay proportional to

$$n^{\frac{\log(2m-1)}{\log(m)}},$$

however, big values of m are never used in practice (the algorithm becomes too complex). A good reference on Toom–Cook multiplication is the paper by Bodrato [38].

7.2.3 FFT-Based Methods

We have seen in the previous section that the idea behind the Toom–Cook family of algorithms consists in reducing a multiplication algorithm to two polynomial evaluations followed by a polynomial interpolation, using evaluation/interpolation points of the form $0, \pm 1, \pm 2, \ldots$.

A. Schönhage and V. Strassen [413] noticed that if we choose the *roots of unity* (either in the complex field or in some carefully chosen finite ring), we can use a very efficient algorithm for evaluating a polynomial at these points: the *Fast Fourier Transform* (FFT) [108]. They introduce two algorithms: an algorithm based on the FFT in the complex field that multiplies two n-bit numbers in time[11] $n \ln n \ln\ln n \ln\ln\ln n \cdots 2^{\mathcal{O}\log^*(n)}$, and an algorithm based on the FFT in the ring of integers modulo $2^{2^m} + 1$, that multiplies two n-bit numbers in time $\mathcal{O}(n \ln n \ln\ln n)$. Another, asymptotically faster, FFT-based multiplication algorithm, of complexity $n \ln(n) 2^{\mathcal{O}(\log^*(n))}$ was suggested by Fürer [197], and optimized by Harvey, Van der Hoeven, and Lecerf [226]. Schönhage and Strassen's algorithms are the fastest algorithms that are actually implemented in significantly distributed tools.

FFT-based multiplications require significantly more memory than Karatsuba-like methods. As noted by Bailey and Borwein [49], in several of his record-breaking computations of π, Kanada used a Karatsuba method at the highest precision levels, and switched to a FFT-based scheme only when the recursion reached a precision level for which there was enough memory to use FFTs.

In the following, we show how to build several functions upon multiple-precision multiplication, and we call $M(n)$ the delay of n-bit multiplication.

7.3 Multiple-Precision Division and Square-Root

7.3.1 The Newton–Raphson Iteration

The Newton–Raphson iteration (NR for short) is a well-known and very efficient technique for finding roots of functions. It was introduced by Newton around 1669 [361], who improved an older method due to Vieta, to solve polynomial equations (without an explicit use of the derivative). It was then

[11] From n, compute $\ln(n)$, then $\ln(\ln(n))$, then $\ln(\ln(\ln(n)))$, etc., until you get a result less than one. The number of iterations is $\log^* n$.

transformed into an iterative scheme by Raphson a few years later [419]. The modern presentation, with derivatives and possible application to nonpolynomial equations is due to Simpson. The interested reader can consult references [155, 479]. NR-based division and/or square-root have been implemented on many recent processors [331, 333, 366, 391, 406].

Assume we want to compute a root α of some function φ. The NR iteration consists in building a sequence

$$x_{n+1} = x_n - \frac{\varphi(x_n)}{\varphi'(x_n)}. \tag{7.6}$$

If φ has a continuous derivative and if α is a single root (that is, $\varphi(\alpha) = 0$ and $\varphi'(\alpha) \neq 0$), then the sequence converges quadratically to α, provided that x_0 is close enough to α: notice that *global convergence* (i.e., for any x_0) is not guaranteed.

Interestingly enough, the classical NR iteration for evaluating square-roots,

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right),$$

obtained by choosing $\varphi = x^2 - a$, goes back to very ancient times. Al-Khwarizmi mentions this method in his arithmetic book [123]. Moreover, it was already used by Heron of Alexandria (this is why it is frequently quoted as "Heron iteration"), and seems to have been known by the Babylonians 2000 years before Heron [195].

The NR iteration is frequently used for evaluating some arithmetic and algebraic functions. For instance

- By choosing
$$\varphi(x) = \frac{1}{x} - a$$
one gets
$$x_{n+1} = x_n(2 - ax_n).$$
This sequence goes to $1/a$: hence it can be used for computing reciprocals and performing divisions.
- As said above, by choosing
$$\varphi(x) = x^2 - a$$
one gets
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$
This sequence goes to $\sqrt{a}$. Note that this iteration requires a division, an operation that is more expensive than a multiplication, and thus often avoided.
- By choosing
$$\varphi(x) = \frac{1}{x^2} - a$$
one gets
$$x_{n+1} = \frac{x_n}{2}\left(3 - ax_n^2\right).$$

This sequence goes to $1/\sqrt{a}$. It is also frequently used to compute $\sqrt{a}$, obtained by multiplying the final result by a.

A very interesting property of the NR iteration is that it is "self-correcting": a small error in the computation of x_n does not change the value of the limit. This makes it possible to start the iterations with a very small precision, and to double the precision with each iteration. A consequence of this is that under reasonable hypotheses[12], the complexity of square root evaluation or division is the same as that of multiplication: the time required to get $n = 2^k$ bits of a quotient is

$$\begin{aligned}&\mathcal{O}\left(M(1) + M(2) + M(4) + M(8) + \cdots + M\left(2^{k-1}\right) + M\left(2^k\right)\right)\\ &= \mathcal{O}\left(M\left(2^k\right)\right) = \mathcal{O}(M(n)).\end{aligned}$$

Of course there is a hidden constant in the "$\mathcal{O}$", but it is not too large. In practice, the cost of an NR-based division is the cost of a very few multiplications.

7.4 Algorithms Based on the Evaluation of Power Series

When a rather moderate precision (say, up to a few hundreds of bits) is at stake, power series are frequently used to approximate functions (it is of course not possible to dynamically generate minimax or least-squares polynomials for all possible precisions, therefore, the methods of Chapter 3 cannot be used).

For instance, in Bailey's MPFUN library [15], the evaluation of $\exp(x)$ is done as follows.

- **range reduction** We compute
$$r = \frac{x - n \times \ln(2)}{256}$$
where n is an integer, chosen such that
$$-\frac{\ln(2)}{512} \leq r \leq \frac{\ln(2)}{512};$$
- **Taylor approximation**
$$\exp(x) = \left(1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \frac{r^4}{4!} + \ldots\right)^{256} \times 2^n = \exp(r)^{256} \times 2^n,$$
where elevating $\exp(r)$ to the power 256 only requires 8 consecutive squarings, since
$$a^{256} = \left(\left(\left(\left(\left(\left(\left(a^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2; \tag{7.7}$$

[12] It is assumed that there exist two constants α and β, $0 < \alpha, \beta < 1$, such that the delay $M(n)$ of n-bit multiplication satisfies $M(\alpha n) \leq \beta M(n)$ if n is large enough [57]. This assumption is satisfied by the grammar school multiplication method, as well as by the Karatsuba-like, the Toom–Cook, and the FFT-based algorithms.

whereas the logarithm of a is computed using the previous algorithm and the Newton iteration (7.6) with $\varphi(x) = \exp(x) - a$:

$$x_{n+1} = x_n + \frac{a - \exp(x_n)}{\exp(x_n)}.$$

This gives

$$x_n \to \ln(a).$$

7.4.1 Binary Splitting Techniques

It is worth being noticed that a technique called *binary splitting* [61, 207, 212], that consists in carefully adjusting the sizes of all the various terms that occur in a calculation, may allow one to evaluate series up to a very large precision. For instance, to approximate $\exp(x)$ by its order-n Taylor expansion

$$1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}, \tag{7.8}$$

Smith [427], decomposes the calculation as follows:

$$\begin{array}{rcccccc}
\exp(x) = & 1 & + & \dfrac{x^j}{j!} & + & \dfrac{x^{2j}}{(2j)!} & + \cdots \\
+ x & \left[\quad 1 \right. & + & \dfrac{x^j}{(j+1)!} & + & \dfrac{x^{2j}}{(2j+1)!} & \left. + \cdots \right] \\
+ x^2 & \left[\quad \dfrac{1}{2!} \right. & + & \dfrac{x^j}{(j+2)!} & + & \dfrac{x^{2j}}{(2j+2)!} & \left. + \cdots \right] \\
\vdots & & & \vdots & & \vdots & \vdots \\
+ x^{j-1} & \left[\quad \dfrac{1}{(j-1)!} \right. & + & \dfrac{x^j}{(2j-1)!} & + & \dfrac{x^{2j}}{(3j-1)!} & \left. + \cdots \right].
\end{array} \tag{7.9}$$

The first step of the algorithm consists of evaluating the j partial sums

$$S_k = \frac{1}{k!} + \frac{x^j}{(j+k)!} + \frac{x^{2j}}{(2j+k)!} + \cdots \tag{7.10}$$

that correspond to the rows in (7.9). In order to do that, each term $x^{ij}/(ij+k)!$, for $k \neq 0$, is deduced from the term $x^{ij}/(ij+k-1)!$ in the row immediately above by the means of a (cheap—its cost is linear in the size of the largest operand) division by the small integer k, and accumulated in the partial sum that will deliver S_k. If $k = 0$, $x^{ij}/(ij+k)!$ is deduced from the last term, $x^{(i-1)j}/(ij-1)!$, of the previous column in (7.9), by the means of a multiplication by the (precomputed) term x^j and a (cheap) division by the small integer ij. Hence, adding a new term to each partial sum S_k in (7.10) needs one multiplication by x^j, j divisions by a small integer, and j additions. A fast exponentiation algorithm (see for instance [275]) allows one to evaluate x^j by performing at most $2\log_2 j - 1$ multiplications. The final result is obtained by evaluating, using Horner's scheme, the polynomial

$$\exp(x) \approx S_0 + x(S_1 + x(S_2 + \cdots + x(S_{j-2} + xS_{j-1})\cdots)).$$

To evaluate the order-n Taylor expansion (7.8), this scheme needs $\mathcal{O}(n/j + j)$ multiplications, plus $\mathcal{O}(n)$ operations whose cost is comparable to the cost of an addition. Smith deduces the time required to evaluate the exponential of y with around t bits of accuracy as follows. If from the initial value y in some bounded interval we use a formula of the form (7.7) to reduce the argument to a number x of absolute value less than 2^{-k}, we will need to perform $\mathcal{O}(k)$ multiplications. The value of n such that the order-n Taylor expansion approximates e^x with accuracy better than 2^{-t} satisfies

$$\frac{(2^{-k})^n}{n!} \approx 2^{-t},$$

which gives (using Stirling's formula[13])

$$n \approx \frac{t \ln(2)}{\ln(t) + k \ln(2)}.$$

Therefore, the total number of multiplications to be performed is

$$\mathcal{O}\left(\frac{t \ln(2)}{j(\ln(t) + k \ln(2))} + j + k\right).$$

Choosing j and k that minimize this gives $j \approx t^{1/3}$ and $k \approx t^{1/3} - \ln(t)/\ln(2)$, which results in $\mathcal{O}(t^{1/3})$ multiplications. Therefore, using Smith's algorithm, computing around t bits of $\exp(y)$ is done in time $\mathcal{O}(t^{1/3} M(t))$.

Johansson [257, 258] uses an improved version of Smith's splitting strategy [427] to provide very fast elementary functions for the medium precision range (up to approximately 4096 bits).

7.5 The Arithmetic–Geometric Mean (AGM)

7.5.1 Presentation of the AGM

When a very high precision is required[14], Taylor series are no longer of interest, and we must use the *arithmetic–geometric mean* [50], defined as follows. Given two positive real numbers a_0 and b_0, one can easily show that the two sequences (a_n) and (b_n) defined below have a common limit, and that their convergence is quadratic

$$\begin{cases} a_{n+1} = \dfrac{a_n + b_n}{2} \\ b_{n+1} = \sqrt{a_n b_n}. \end{cases}$$

Define $A(a_0, b_0)$ as the common limit of these sequences, $A(a_0, b_0)$ is called the *arithmetic–geometric mean* (AGM for short) of a_0 and b_0. Gauss noticed that $A(1, x)$ is equal to

$$\frac{\pi}{2F(x)},$$

[13] $n! \approx \sqrt{2n\pi}(n/e)^n$.

[14] According to Borwein and Borwein [51], the switchover is somewhere in the 100 to 1000 decimal digit range.

where F is the elliptic function:

$$F(x) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-(1-x^2)\sin^2\theta}}.$$

It is worth noticing that the AGM iteration is not self-correcting: we cannot use the trick of doubling the precision at each iteration, that we use with the Newton–Raphson iterations. All iterations must be performed with full precision.

7.5.2 Computing Logarithms with the AGM

The AGM gives a fast way of computing function $F(x)$. This might seem of little use, but now, if we notice that

$$\begin{aligned} F(4/s) = \ln(s) &+ \frac{4\ln(s)-4}{s^2} \\ &+ \frac{36\ln(s)-42}{s^4} + \frac{1200\ln(s)-1480}{3s^6} \\ &+ \mathcal{O}\left(\frac{1}{s^8}\right), \end{aligned} \tag{7.11}$$

we can see that if s is "large enough", then $F(4/s)$ is a good approximation to $\ln(s)$. One can easily see what "large enough" means here: if we wish around p bits of accuracy in the result, then the second term of the series (7.11) must be at least 2^p times smaller than the leading term, i.e., s^2 must be larger than 2^p. To evaluate $\ln(x)$ with around p bits of accuracy, we will therefore first compute a real number s and an integer m such that $s = x2^m > 2^{p/2}$. Then, $\ln(x)$ will be obtained as

$$\ln(x) \approx \frac{\pi}{2A(1,4/s)} - m\ln(2). \tag{7.12}$$

Example: computation of $\ln(25)$.

Assume that we wish to evaluate $\ln(25)$ with around 1000 bits of accuracy (i.e., slightly more than 301 decimal digits), using the previous method.

The smallest integer m such that

$$25 \times 2^m > 2^{500}$$

is 496. Therefore, we choose

$$\begin{cases} m = 496 \\ s = 25 \times 2^{496}. \end{cases}$$

We therefore have to compute $A(1, 4/s)$ using the AGM iteration, with

$$
\begin{aligned}
4/s = {} & 7.82063709055898798605306724662686131146087101586515625956765160776010656390328437171675452187651898433760\\
& 0390895591595917613704775178966962521940772354349797454654516566458501688411107378001622274090262189901259\\
& 4890589782782388544418943460421174272902274101535208638673974261798262712603991118844289965640614870\cdots\\
& \times 10^{-151}
\end{aligned}
$$

After 16 iterations, we get

$$
\begin{aligned}
A(1, 4/s) = {} & 0.00452653127147250096072983021051330291298273470158382817131216067448690439957375794792338578019057417700371454676009364074131526473035830857845058014797497352119203403501958066084502701791872357638470143804008606271554404074490685737500078687218841447201069157798369164678109017756105713418996572318763110465033 9\cdots
\end{aligned}
$$

Hence,

$$
\begin{aligned}
\ln(25) &\approx \tfrac{\pi}{2A(1,4/s)} - m\ln(2)\\
&\approx 3.21887582486820074920151866645237527905120270853703544382529578294835797541531552926026775618635922159993260604343112579944801045864935239926723323492741145510435927499436649130698571240468305011454031038720175955477945137638708142553230946244361905589704258564271611944513534457057448092317889635678225114219\cdots
\end{aligned}
$$

All the digits displayed above but the last nine coincide with those of the decimal representation of $\ln(25)$. The obtained relative error is $2^{-992.5}$.

Computation of π and $\ln(2)$.

Using (7.12) requires the knowledge of at least p bits of π and $\ln(2)$. For small values of p, these bits will be precomputed and stored, but for large values, they will be dynamically computed. Concerning $\ln(2)$ we can directly use (7.11): if we want around p bits of accuracy, we approximate $\ln(2^{p/2}) = p\ln(2)/2$ by $F(2^{-p/2+2})$ using (7.11) and the AGM iteration. This gives

$$
\ln(2) \approx \frac{\pi}{pA(1, 2^{-p/2+2})}. \tag{7.13}
$$

Table 7.1 *The first terms of the sequence p_k generated by the Brent–Salamin algorithm. That sequence converges to π quadratically.*

k	p_k
1	3.187672642712108627201929970525369232651053571859369226487 63
2	3.141680293297653293918070424560009382795719438815402832644 19
3	3.141592653895446496002914758818043486108879237261311589651 10
4	3.141592653589793238466360602706631321757702411342429356486 85
5	3.141592653589793238462643383279502884197169949164726605834 70
6	3.141592653589793238462643383279502884197169399375105820974 94

There are several ways of computing π [21]. We can for instance use the following algorithm, due to Brent [57] and Salamin [408], based on the AGM:

$$\begin{aligned} a_0 &= 1 \\ b_0 &= \frac{1}{\sqrt{2}} \\ s_0 &= \frac{1}{2} \\ a_k &= \frac{a_{k-1} + b_{k-1}}{2} \\ b_k &= \sqrt{a_{k-1} b_{k-1}} \\ s_k &= s_{k-1} - 2^k (a_k^2 - b_k^2) \\ p_k &= \frac{2a_k^2}{s_k}. \end{aligned}$$

The sequence p_k, whose first terms are given in Table 7.1, converges quadratically to π.

7.5.3 Computing Exponentials with the AGM

The exponential of x is computed using the previous algorithm for evaluating logarithms and the Newton–Raphson iteration. More precisely, to compute $\exp(a)$, we use iteration (7.6) with $\varphi(x) = \ln(x) - a$. This gives

$$x_{n+1} = x_n \left(1 + a - \ln(x_n)\right), \tag{7.14}$$

where $\ln(x_n)$ is evaluated using the method given in Section 7.5.2. An example (computation of $\exp(1)$) is given in Table 7.2. A fast and easy way of generating a seed value x_0 close to $\exp(a)$ is to compute in conventional (single- or double-precision) floating-point arithmetic an approximation to the exponential of the floating-point number that is nearest a.

7.5.4 Very Fast Computation of Trigonometric Functions

The methods presented in the previous sections, based on the AGM, for computing logarithms and exponentials, can be extended to trigonometric functions (which is not surprising: (7.11) remains true if s is a complex number). Several variants have been suggested for these functions. Here is Brent's

Table 7.2 *First terms of the sequence x_n generated by the NR iteration for computing* $\exp(a)$, *given here with* $a = 1$ *and* $x_0 = 2.718$. *The sequence goes to e quadratically.*

n	x_n
1	2.71828181384870854839394204546332554936588598573150616522 ···
2	2.71828182845904519609615815000766898668384995737383488643 ···
3	2.71828182845904523536028747135266221418255751906243823415 ···
4	2.71828182845904523536028747135266249775724709369995957496 ···
5	2.71828182845904523536028747135266249775724709369995957496 ···

algorithm for computing $\arctan(x)$, with $0 < x \leq 1$, to approximately p bits of precision. We first start with

$$\begin{cases} s_0 = 2^{-p/2} \\ v_0 = x/(1+\sqrt{1+x^2}) \\ q_0 = 1 \end{cases}$$

and we iterate:

$$\begin{cases} q_{i+1} = 2q_i/(1+s_i) \\ a_i = 2s_i v_i/(1+v_i^2) \\ b_i = a_i/\left(1+\sqrt{1-a_i^2}\right) \\ c_i = (v_i+b_i)/(1-v_i b_i) \\ v_{i+1} = c_i/\left(1+\sqrt{1+c_i^2}\right) \\ s_{i+1} = 2\sqrt{s_i}/(1+s_i) \end{cases}$$

until $1 - s_j \leq 2^{-p}$. We then have

$$\arctan(x) \approx q_j \ln\left(\frac{1+v_j}{1-v_j}\right).$$

The convergence is quadratic, and using this algorithm we can evaluate n bits of an arctangent in time $\mathcal{O}(M(n)\ln n)$ [57]. Using the Newton–Raphson iteration, we can now evaluate the tangent function: assume we wish to compute $\tan(\theta)$. With $\varphi(x) = \arctan(x) - \theta$, iteration (7.6) becomes

$$x_{n+1} = x_n - \left(1+x_n^2\right)(\arctan(x_n) - \theta). \tag{7.15}$$

The sequence x_n converges to $\tan(\theta)$ quadratically provided that x_0 is close enough to $\tan(\theta)$. Again, a good seed value x_0 is easily obtained by computing $\tan(\theta)$ in conventional floating-point arithmetic. Using the "self-correcting" property of the Newton–Raphson iteration, as we did with division, we double the working precision at each iteration, performing the last iteration only with full precision, so that computing a tangent is done in time $\mathcal{O}(M(n)\ln n)$ [57], i.e., with the same complexity as the arctangent function. As suggested by Brent, $\sin(\theta)$, and $\tan(\theta)$ can then be computed using

Table 7.3 *Time complexity of the evaluation of some functions in multiple-precision arithmetic (extracted from Table 7.1 of [51]). $M(n)$ is the complexity of n-bit multiplication.*

function	complexity
addition	$\mathcal{O}(n)$
multiplication	$\mathcal{O}(n\ln(n)\ln\ln(n)))$
division, sqrt	$\mathcal{O}(M(n))$
ln, exp	$\mathcal{O}(M(n)\ln(n))$
sin, cos, arctan	$\mathcal{O}(M(n)\ln(n))$

$$\begin{aligned} x &= \tan\left(\frac{\theta}{2}\right) \\ \sin(\theta) &= \frac{2x}{1+x^2} \\ \cos(\theta) &= \frac{1-x^2}{1+x^2}. \end{aligned}$$

There are similar algorithms for many usual functions. Table 7.3 gives the time complexity for the evaluation of the most common ones.

Part II
Shift-and-Add Algorithms

Introduction to Shift-and-Add Algorithms 8

At the beginning of the seventeenth century, there was a terrible food shortage. To curb it, the King decided to implement a tax on bread consumption. The tax would be proportional to the exponential of the weight of the bread! Bakers and mathematicians had the same question in mind: how could they quickly compute the price of bread? An old mathematician, called Briggs, found a convenient solution. He said to the King,

> "To calculate the tax, all I need is a pair of scales and a file."

Rather surprised, the King nevertheless asked his servants to give him the required material. First, Briggs spent some time filing the different weights of the pair of scales. (Table 8.1 gives the weight of the different weights after the filing). Then he said,

> "Give me a loaf of bread."

He weighed the loaf of bread, and found an *apparent* weight — remember, the weights were filed ! — equal to 0.572 pounds. Then he said,

> "I write 0.572; I replace the leading zero by a one; this gives 1.572. Now I calculate the product of the first two fractional digits (i.e., 5×7), and I divide this by $1,000$. This gives 0.035. I calculate the product of the first and the third fractional digits (i.e., 5×2), and I divide this by $10,000$. This gives 0.001. I add 0.035 and 0.001 to 1.572, and I obtain the exponential of the weight, which is 1.608."

The King was rather skeptical. He asked his servants to find the actual weight of the bread (using unfiled weights!), and they came up with 0.475 pounds. Next he asked his mathematicians to compute the exponential of 0.475. After long calculations, they found the result: $1.608014\cdots$ Briggs' estimation was not too bad!

Let us explain how Briggs' method worked: first, the weights were filed so that a weight of x pounds actually weighed $\ln(1+x)$ pounds after the filing. Therefore if the "apparent" weight of the bread was, say, $0.x_1x_2x_3$ pounds, then its real weight was:

$$\ln\left(1+\frac{x_1}{10}\right)+\ln\left(1+\frac{x_2}{100}\right)+\ln\left(1+\frac{x_3}{1,000}\right)$$

pounds, the exponential of which is:

$$\left(1+\frac{x_1}{10}\right)\left(1+\frac{x_2}{100}\right)\left(1+\frac{x_3}{1,000}\right)$$
$$\simeq 1+\frac{x_1}{10}+\frac{x_2}{100}+\frac{x_3}{1,000}+\frac{x_1x_2}{1,000}+\frac{x_1x_3}{10,000}.$$

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_8

Table 8.1 *The filing of the different weights.*

Original Weight	Weight After Filing
0.5	0.405
0.4	0.336
0.3	0.262
0.2	0.182
0.1	0.095
0.09	0.086
0.08	0.077
0.07	0.068
0.06	0.058
0.05	0.048
0.04	0.039
less than 0.03	unchanged

Although this story is pure fiction (it was invented by Xavier Merrheim to defend his Ph.D. dissertation [342]), Henry Briggs (1561–1631) did actually exist. He was a contemporary of Napier (1550–1617, the inventor of the logarithms), and he designed the first convenient algorithms for computing logarithms [411]. He published 14-digit accurate tables in his *Arithmetica Logarithmica* (1624). A fascinating description of the various methods used by Briggs for building his tables is given by Roegel [401].

Briggs' algorithm was the first *shift-and-add* algorithm. Shift-and-add algorithms allow the evaluation of elementary functions using very simple elementary operations: *addition*, *multiplication by a power of the radix of the number system being used* (in fixed-point arithmetic, such multiplications are performed by the means of simple shifts), and *multiplication by one radix-r digit*. In this chapter, we present simple shift-and-add algorithms in order to introduce basic notions that are used in the following chapters. This class of algorithms is interesting mainly for hardware implementations.

8.1 The Restoring and Nonrestoring Algorithms

Now let us examine a very simple algorithm (quite close to Briggs' algorithm) to exhibit the properties that make it work.

Algorithm 16 Algorithm exponential 1

input values: t, N (N is the number of steps)
output value: E_N
define

$$\begin{aligned} t_0 &= 0 \\ E_0 &= 1; \end{aligned}$$

build two sequences t_n and E_n as follows

$$\begin{aligned} t_{n+1} &= t_n + d_n \ln\left(1 + 2^{-n}\right) \\ E_{n+1} &= E_n\left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n} \\ d_n &= \begin{cases} 1 \text{ if } t_n + \ln\left(1 + 2^{-n}\right) \leq t \\ 0 \text{ otherwise.} \end{cases} \end{aligned} \tag{8.1}$$

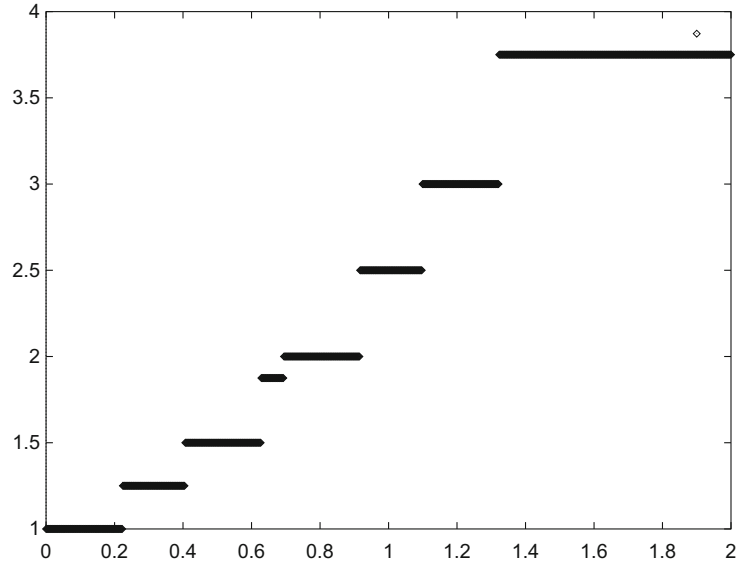


Figure 8.1 *Value of* E_3 *versus* t.

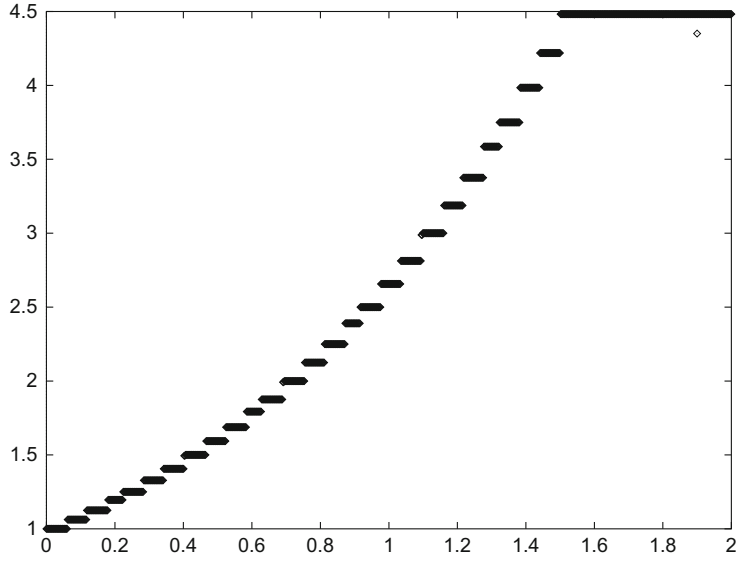


Figure 8.2 *Value of* E_5 *versus* t.

This algorithm only requires additions and multiplications by powers of 2. Such multiplications reduce to shifts when implemented in radix-2 arithmetic. Figures 8.1, 8.2 and 8.3 plot the values of E_3, E_5, and E_{11} versus t.

By examining those figures, one can see that there is an interval $I \approx [0, 1.56]$ in which E_n seems to converge toward a regular function of t as n goes toward infinity. *Let us temporarily admit that this function is the exponential function*. One can easily verify that E_n is always equal to e^{t_n}. Therefore, a consequence of our temporary assumption is

$$\lim_{n \to +\infty} t_n = t. \tag{8.2}$$

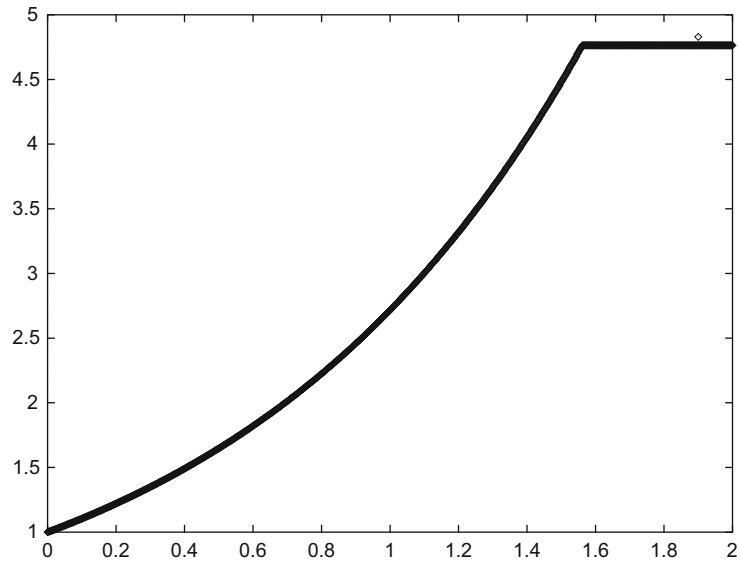


Figure 8.3 *Value of* E_{11} *versus* t.

Since t_n is obviously equal to $\sum_{i=0}^{n-1} d_n \ln\left(1+2^{-n}\right)$, this implies

$$t = \sum_{i=0}^{\infty} d_n \ln\left(1+2^{-n}\right). \tag{8.3}$$

Thus it seems that our algorithm makes it possible to decompose any number t belonging to I into a sum

$$t = d_0 w_0 + d_1 w_1 + d_2 w_2 + \cdots,$$

where $w_i = \ln\left(1+2^{-i}\right)$. Now let us try to characterize sequences (w_i) such that similar decompositions are possible, and to find algorithms that lead to those decompositions. We already know one such sequence: if $w_i = 2^{-i}$, then the selection of the digits of the binary expansion of t for the d_is provides a decomposition of any $t \in [0, 2)$ into a sum of $d_i w_i$s.

The following theorem gives an algorithm for computing such a decomposition, provided that the sequence (w_i) satisfies some simple properties. This algorithm can be viewed as equivalent to the weighing of t using a pair of scales, and weights (the terms w_i) that are either unused or put in the pan that does not contain t. Figure 8.4 illustrates this.

Theorem 16 (restoring decomposition algorithm). *Let* (w_n) *be a decreasing sequence of positive real numbers such that the power series* $\sum_{i=0}^{\infty} w_i$ *converges. If*

$$\forall n,\ w_n \leq \sum_{k=n+1}^{\infty} w_k, \tag{8.4}$$

then for any $t \in \left[0, \sum_{k=0}^{\infty} w_k\right]$, *the sequences* (t_n) *and* (d_n) *defined as*

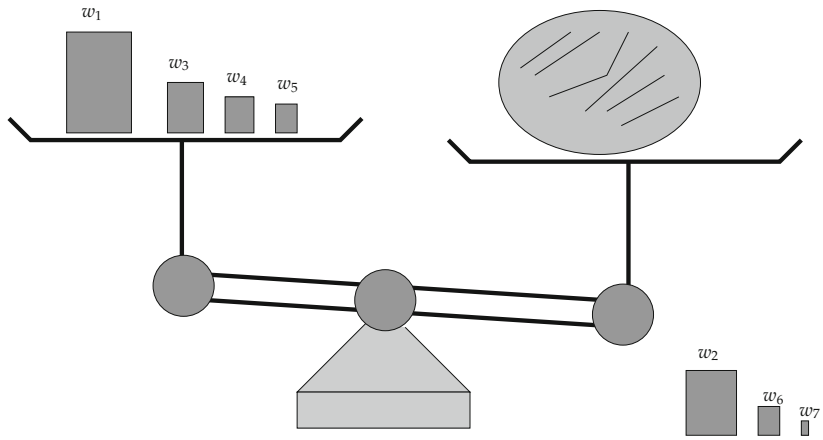


Figure 8.4 *The restoring algorithm. The weights are either unused or put on the pan that does not contain the loaf of bread being weighed. In this example, the weight of the loaf of bread is* $w_1 + w_3 + w_4 + w_5 + \cdots$.

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 \; if \;\; t_n + w_n \leq t \\ 0 \; otherwise \end{cases} \end{aligned} \tag{8.5}$$

satisfy

$$t = \sum_{n=0}^{\infty} d_n w_n = \lim_{n \to \infty} t_n.$$

Proof of Theorem 16
Let us prove by induction that for any n,

$$0 \leq t - t_n \leq \sum_{k=n}^{\infty} w_k. \tag{8.6}$$

Since the power series $\sum_{i=0}^{\infty} w_i$ converges, the remainder $\sum_{k=n}^{\infty} w_k$ goes to zero as n goes to infinity; therefore proving (8.6) would suffice to prove the theorem. Relation (8.6) is obviously true for $n = 0$. Let us assume it is true for some n.

- If $d_n = 0$, then $t_{n+1} = t_n$, and t_n satisfies $t_n + w_n > t$. Therefore $0 \leq t - t_{n+1} = t - t_n < w_n$. Using (8.4), we get $0 \leq t - t_{n+1} \leq \sum_{k=n+1}^{\infty} w_k$.
- If $d_n = 1$, then $t_{n+1} = t_n + w_n$; therefore $t - t_{n+1} \leq \sum_{k=n}^{\infty} w_k - w_n = \sum_{k=n+1}^{\infty} w_k$. Moreover (after the choice of d_n), $t_n + w_n \leq t$; therefore $t - t_{n+1} = t - t_n - w_n \geq 0$, q.e.d.

A sequence (w_n) that satisfies the conditions of Theorem 16 is called a *discrete base* [352, 353, 354].

The algorithm given by (8.5) is called here the "restoring algorithm", by analogy with the restoring division algorithm [179] (if we choose $w_i = y2^{-i}$, we get the restoring division algorithm; we obtain $t = \sum_{i=0}^{\infty} d_i y 2^{-i}$, which gives $t/y = d_0.d_1 d_2 d_3 \cdots$). To our knowledge, this analogy between some division algorithms and most shift-and-add elementary function algorithms was first pointed out by

Meggitt [337], who presented algorithms similar to those in this chapter as "pseudomultiplication" and "pseudodivision" methods. Other "pseudodivision" algorithms were suggested by Sarkar and Krishnamurthy [407]. An algorithm very similar to the restoring exponential algorithm was suggested by Chen [79]. The following theorem presents another algorithm that gives decompositions with values of the $d_i s$ equal to -1 or $+1$. This algorithm, called the "nonrestoring algorithm" by analogy with the nonrestoring division algorithm, is used in the CORDIC algorithm (see Chapter 9).

Theorem 17 (nonrestoring decomposition algorithm). *Let (w_n) be a decreasing sequence of positive real numbers such that $\sum_{i=0}^{\infty} w_i$ converges. If*

$$\forall n, w_n \leq \sum_{k=n+1}^{\infty} w_k, \tag{8.7}$$

then for any $t \in \left[-\sum_{k=0}^{\infty} w_k, \sum_{k=0}^{\infty} w_k\right]$, the sequences (t_n) and (d_n) defined as

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 \text{ if } t_n \leq t \\ -1 \text{ otherwise} \end{cases} \end{aligned} \tag{8.8}$$

satisfy

$$t = \sum_{n=0}^{\infty} d_n w_n = \lim_{n \to \infty} t_n.$$

The proof of this theorem is very similar to the proof of Theorem 16. The nonrestoring algorithm can be viewed as the weighing of t using a pair of scales, and weights (the terms w_i) which must be put in one of the pans (no weight is unused). Figure 8.5 illustrates this.

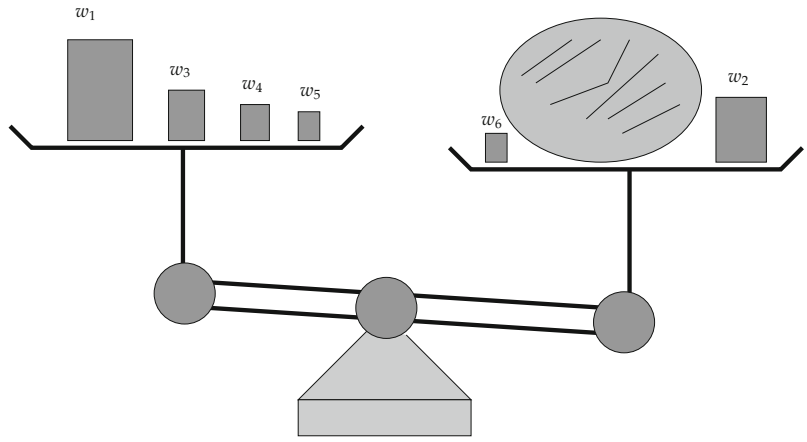


Figure 8.5 *The nonrestoring algorithm. All the weights are used, and they can be put on both pans. In this example, the weight of the loaf of bread is $w_1 - w_2 + w_3 + w_4 + w_5 - w_6 + \cdots$.*

8.2 Simple Algorithms for Exponentials and Logarithms

8.2.1 The Restoring Algorithm for Exponentials

We admit that the sequences $\ln\left(1+2^{-n}\right)$ and $\arctan 2^{-n}$ are discrete bases, that is, that they satisfy the conditions of Theorems 16 and 17 (see [352] for proof). Now let us again find Algorithm 8.1 using Theorem 16. We use the discrete base $w_n = \ln\left(1+2^{-n}\right)$. Let $t \in \left[0, \sum_{k=0}^{\infty} w_k\right] \approx [0, 1.56202\cdots]$. From Theorem 16, the sequences (t_n) and (d_n) defined as

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n \ln\left(1+2^{-n}\right) \\ d_n &= \begin{cases} 1 \text{ if } t_n + \ln\left(1+2^{-n}\right) \leq t \\ 0 \text{ otherwise} \end{cases} \end{aligned}$$

satisfy

$$t = \sum_{n=0}^{\infty} d_n \ln\left(1+2^{-n}\right) = \lim_{n\to\infty} t_n.$$

Now let us try to build a sequence E_n such that at any step n of the algorithm,

$$E_n = \exp\left(t_n\right). \tag{8.9}$$

Since $t_0 = 0$, E_0 must be equal to 1. When t_{n+1} is different from t_n (i.e., when $d_n = 1$), t_{n+1} is equal to $t_n + \ln\left(1+2^{-n}\right)$. To keep relation (8.9) invariable, one must multiply E_n by $\exp\ln\left(1+2^{-n}\right) = \left(1+2^{-n}\right)$. Since t_n goes to t, E_n goes to e^t, and we find Algorithm 8.1 again.

It is worth noticing that when building this algorithm, we never really used the fact that the logarithms that appear are *natural* (i.e., base-e) logarithms. If we replace, in the algorithm, the constants $\ln\left(1+2^{-n}\right)$ by $\log_a\left(1+2^{-n}\right)$, then we obtain an algorithm for computing a^t. Its convergence domain is

$$[0, \sum_{n=0}^{\infty} \log_a\left(1+2^{-n}\right)].$$

Error evaluation

Let us estimate the error on the result E_n if we stop at step n of the algorithm, that is, if we approximate the exponential of t by E_n. From the proof of Theorem 16, we have $0 \leq t - t_n \leq \sum_{k=n}^{\infty} \ln\left(1+2^{-k}\right)$. Since $\ln(1+x) < x$ for any $x > 0$, we have

$$\sum_{k=n}^{\infty} \ln\left(1+2^{-k}\right) \leq \sum_{k=n}^{\infty} 2^{-k} = 2^{-n+1}.$$

Therefore $0 \leq t - t_n \leq 2^{-n+1}$. Now from $E_n = \exp\left(t_n\right)$ we deduce:

$$1 \leq \exp\left(t - t_n\right) \leq \exp\left(2^{-n+1}\right).$$

This gives

$$\left|\frac{e^t - E_n}{e^t}\right| \leq 1 - e^{-2^{-n+1}} \leq 2^{-n+1}. \tag{8.10}$$

Therefore, when stopping at step n, the *relative error* on the result[1] is bounded by 2^{-n+1} (i.e., we roughly have $n-1$ significant bits).

8.2.2 The Restoring Algorithm for Logarithms

From the previous algorithm, we can easily deduce another algorithm that evaluates logarithms. Let us assume that we want to compute $\ell = \ln(x)$. First, assuming that ℓ is known, we compute its exponential x using the previously studied algorithm. This gives:

$$\begin{array}{ll} t_0 & = 0 \\ E_1 & = 1 \\ t_{n+1} & = t_n + d_n \ln\left(1 + 2^{-n}\right) \\ E_{n+1} & = E_n + d_n E_n 2^{-n} \end{array} \tag{8.11}$$

with

$$d_n = \begin{cases} 1 \text{ if } t_n + \ln\left(1 + 2^{-n}\right) \leq \ell \\ 0 \text{ otherwise.} \end{cases} \tag{8.12}$$

The previous study shows that, using this algorithm:

$$\begin{array}{ll} \lim_{n\to\infty} t_n & = \ell \\ \lim_{n\to\infty} E_n & = \exp(\ell) = x. \end{array}$$

Of course, this algorithm cannot be used to compute ℓ since (8.12) requires the knowledge of ℓ! However, since the sequence E_n was built such that at any step n, $E_n = \exp(t_n)$, the test performed in (8.12) is equivalent to the test:

$$d_n = \begin{cases} 1 \text{ if } E_n \times \left(1 + 2^{-n}\right) \leq x \\ 0 \text{ otherwise.} \end{cases} \tag{8.13}$$

Consequently, if we replace (8.12) by (8.13) in the previous algorithm, we will get the *same results* (including the convergence of the sequence t_n to ℓ) without now having to know ℓ in advance!. This gives the *restoring logarithm algorithm.*

[1] In this estimation, we do not take the rounding errors into account. They will depend on the precision used for representing the intermediate variables of the algorithm.

Algorithm 17 Restoring logarithm algorithm

- **input values:** x, n (n is the number of steps), with

$$1 \leq x \leq \prod_{i=0}^{\infty} \left(1 + 2^{-i}\right) \approx 4.768;$$

- **output value:** $t_n \approx \ln x$.

Define $t_0 = 0$ and $E_0 = 1$.
Build two sequences t_i **and** E_i **as follows**

$$\begin{array}{lll} t_{i+1} & = t_i + \ln\left(1 + d_i 2^{-i}\right) \\ E_{i+1} & = E_i\left(1 + d_i 2^{-i}\right) = E_i + d_i E_i 2^{-i} \\ d_i & = \begin{cases} 1 \text{ if } E_i + E_i 2^{-i} \leq x \\ 0 \text{ otherwise.} \end{cases} \end{array} \tag{8.14}$$

As in the previous section, if we replace, in the algorithm, the constants $\ln\left(1 + 2^{-n}\right)$ by $\log_a\left(1 + 2^{-n}\right)$, we get an algorithm that evaluates base-a logarithms.

Error evaluation

From the proof of Theorem 16, we have

$$0 \leq \ell - t_n \leq \sum_{k=n}^{\infty} \ln\left(1 + 2^{-k}\right) \leq 2^{-n+1}.$$

Therefore, if we stop the algorithm at step n, the *absolute error* on the result is bounded[2] by 2^{-n+1}.

8.3 Faster Shift-and-Add Algorithms

In the previous sections we studied algorithms that were similar to the restoring and nonrestoring division algorithms. Our aim now is to design faster algorithms, similar to the SRT division algorithms [179, 399, 400], using redundant number systems. The algorithms presented in this section are slight variations of algorithms proposed by N. Takagi in his Ph.D. dissertation [443].

8.3.1 Faster Computation of Exponentials

First, we try to compute exponentials in a more efficient way. Let us start from the basic iteration (8.1). Defining $L_n = t - t_n$ and noticing that

$$d_n \ln\left(1 + 2^{-n}\right) = \ln\left(1 + d_n 2^{-n}\right)$$

for $d_n = 0$ or 1, we get:

[2] In this estimation, we do not take the rounding errors into account. They will depend on the precision used for representing the intermediate variables of the algorithm.

$$\begin{aligned} L_{n+1} &= L_n - \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 2^{-n}\right) \\ d_n &= \begin{cases} 1 \text{ if } L_n \geq \ln\left(1 + 2^{-n}\right) \\ 0 \text{ otherwise.} \end{cases} \end{aligned} \tag{8.15}$$

If $L_0 = t$ is less than $\sum_{n=0}^{\infty} \ln\left(1 + 2^{-n}\right)$, this gives:

$$\begin{aligned} L_n &\to 0 \\ n &\to +\infty. \end{aligned}$$

and

$$\begin{aligned} E_n &\to E_0 e^{L_0} \\ n &\to +\infty. \end{aligned}$$

To accelerate the computation, we try to perform this iteration using a redundant (e.g., carry-save or signed-digit) number system (see Chapter 2). The additions that appear in this iteration are quickly performed, without carry propagation. Unfortunately, the test "$L_n \geq \ln\left(1 + 2^{-n}\right)$" may require the examination of a number of digits that may be close to the word length.[3] This problem is solved by adding some "redundancy" to the algorithm. Instead of only allowing the values $d_i = 0, 1$, we also allow $d_i = -1$. Since $\ln\left(1 - 2^{-0}\right)$ is not defined, we must start at step $n = 1$. To ensure that the algorithm converges, we must find values d_n such that L_n will go to zero. This can be done by arranging that $L_n \in [s_n, r_n]$, where

$$\begin{aligned} r_n &= \sum_{k=n}^{\infty} \ln\left(1 + 2^{-k}\right) \\ s_n &= \sum_{k=n}^{\infty} \ln\left(1 - 2^{-k}\right). \end{aligned} \tag{8.16}$$

Figure 8.6 presents the different possible values of L_{n+1} versus L_n (following (8.15)) depending on the choice of d_n. This figure is very close to the *Robertson diagrams* that appear in the SRT division algorithm [179, 399], so we call it the *Robertson diagram* of iteration (8.15). Assume that $L_n \in [s_n, r_n]$. We want L_{n+1} to be in $[s_{n+1}, r_{n+1}]$. The diagram shows that we may select:

- $d_n = -1$ if $L_n \leq \overline{A}_n$;
- $d_n = 0$ if $\overline{B}_n \leq L_n \leq B_n$;
- $d_n = 1$ if $L_n \geq A_n$.

The values A_n, B_n, $\overline{A}_n$, and $\overline{B}_n$ of Figure 8.6 satisfy:

$$\begin{cases} A_n = s_{n+1} + \ln\left(1 + 2^{-n}\right) \\ B_n = r_{n+1} \\ \overline{A}_n = r_{n+1} + \ln\left(1 - 2^{-n}\right) \\ \overline{B}_n = s_{n+1}. \end{cases} \tag{8.17}$$

[3] If d_n was chosen by means of a comparison, computation would be slowed down — the time saved by performing fully parallel additions would be lost — and huge tables would be required if d_n was given by a table lookup — and, incidentally, if we were able to implement such huge tables efficiently, we would rather directly tabulate the exponential function.

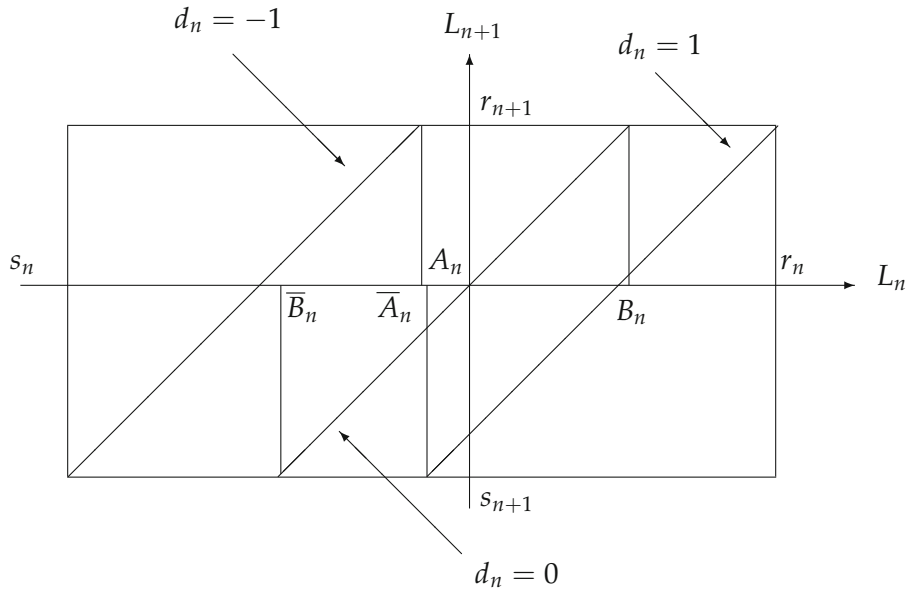


Figure 8.6 *Robertson diagram of the "redundant exponential" algorithm.*

Table 8.2 *First 10 values and limit values of* $2^n s_n$, $2^n r_n$, $2^n A_n$, $2^n B_n$, $2^n \overline{A}_n$, *and* $2^n \overline{B}_n$.

n	$2^n s_n$	$2^n r_n$	$2^n \overline{B}_n$	$2^n \overline{A}_n$	$2^n A_n$	$2^n B_n$
1	−2.484	1.738	−1.098	−0.460	−0.287	0.927
2	−2.196	1.854	−1.045	−0.190	−0.152	0.961
3	−2.090	1.922	−1.022	−0.088	−0.079	0.980
4	−2.043	1.960	−1.011	−0.043	−0.041	0.990
5	−2.021	1.980	−1.005	−0.021	−0.021	0.995
6	−2.011	1.990	−1.003	−0.010	−0.010	0.997
7	−2.005	1.995	−1.001	−0.005	−0.005	0.999
8	−2.003	1.997	−1.001	−0.003	−0.003	0.999
9	−2.001	1.999	−1.000	−0.001	−0.001	1.000
10	−2.001	1.999	−1.000	−0.001	−0.001	1.000
∞	−2	2	−1	0	0	1

Using these relations and the Taylor expansion of the function $\ln(1 + x)$, one can show that:

$$\begin{cases} 2^n A_n \leq & 0 \\ 2^n B_n \geq & 1/2 \\ 2^n \overline{A}_n \geq & -1/2 \\ 2^n \overline{B}_n \leq & -1. \end{cases} \tag{8.18}$$

Table 8.2 gives the first 10 values of $2^n s_n$, $2^n r_n$, $2^n A_n$, $2^n B_n$, $2^n \overline{A}_n$, and $2^n \overline{B}_n$.

We can see that there is an overlap between the area where $d_n = -1$ is a correct choice and the area where $d_n = 0$ is a correct choice. There is another overlap between the area where $d_n = 0$ is a correct choice and the area where $d_n = +1$ is a correct choice. This allows us to choose a convenient value of

d_n by examining a few digits of L_n only. Let us see how this choice can be carried out in signed-digit and carry-save arithmetic.

Signed-digit implementation

Assume we use the signed-digit system. Define L_n^* as $2^n L_n$ *truncated after the first fractional digit*. We have

$$\left|L_n^* - 2^n L_n\right| \leq 1/2.$$

Therefore if we choose[4]

$$d_n = \begin{cases} -1 \text{ if } & L_n^* \leq -1 \\ 0 \text{ if } -1/2 \leq & L_n^* \leq 0 \\ 1 \text{ if } & L_n^* \geq 1/2, \end{cases} \tag{8.19}$$

then L_{n+1} will be between s_{n+1} and r_{n+1}.

Proof

- If $L_n^* \leq -1$, then $2^n L_n \leq -1/2$; therefore $2^n L_n \leq 2^n \overline{A}_n$. This implies $L_n \leq \overline{A}_n$; therefore (see Figure 8.6) choosing $d_n = -1$ will ensure $s_{n+1} \leq L_{n+1} \leq r_{n+1}$.
- If $-1/2 \leq L_n^* \leq 0$, then[5] $-1 \leq 2^n L_n \leq 1/2$; therefore $\overline{B}_n \leq L_n \leq B_n$. Therefore choosing $d_n = 0$ will ensure $s_{n+1} \leq L_{n+1} \leq r_{n+1}$.
- If $L_n^* \geq 1/2$, then $2^n L_n \geq 0$; therefore $L_n \geq A_n$, and choosing $d_n = 1$ will ensure $s_{n+1} \leq L_{n+1} \leq r_{n+1}$.

Therefore *one fractional digit of* L_n^* *suffices to choose a correct value of* d_n. Now let us see how many digits of the *integer part* we need to examine. One can easily show:

$$\begin{cases} -5/2 < 2^n s_n \\ 2^n r_n < 2; \end{cases} \tag{8.20}$$

therefore, for any n,

$$-5/2 < 2^n L_n < 2.$$

Since $|2^n L_n - L_n^*| \leq 1/2$, we get

$$-3 < L_n^* < \frac{5}{2},$$

and, since L_n^* is a multiple of $1/2$,

$$-\frac{5}{2} \leq L_n^* \leq 2.$$

Define $\widehat{L}_n^*$ as the 4-digit number obtained by truncating the digits of L_n^* of a weight greater than or equal to $8 = 2^3$; that is, if

$$L_n^* = \cdots L_{n,4}^* L_{n,3}^* L_{n,2}^* L_{n,1}^* L_{n,0}^* . L_{n,-1}^*,$$

[4]Remember, L_n^* is a multiple of $1/2$; therefore $L_n^* > -1$ implies $L_n^* \geq -1/2$.

[5]Since L_n^* is a multiple of $1/2$, "$-1/2 \leq L_n^* \leq 0$" is equivalent to "$L_n^* \in \{-1/2, 0\}$."

then

$$\widehat{L}_n^* = L_{n,2}^* L_{n,1}^* L_{n,0}^* . L_{n,-1}^*.$$

It is worth noticing that $\widehat{L}_n^*$ and L_n^* may have different signs: for instance, if $L_n^* = \overline{1}101.0$, then L_n^* is negative, whereas $\widehat{L}_n^* = 101.0$ is positive.
We have

- $-8 + 1/2 \leq \widehat{L}_n^* \leq 8 - 1/2$;
- $L_n^* - \widehat{L}_n^*$ is a multiple of 8.

Therefore,

- If $-8+1/2 \leq \widehat{L}_n^* \leq -6$, then the only possibility compatible with $-5/2 \leq L_n^* \leq 2$ is $L_n^* = \widehat{L}_n^* + 8$. This gives $1/2 \leq L_n^* \leq 2$. Therefore $d_n = 1$ is a correct choice.
- If $-6 + 1/2 \leq \widehat{L}_n^* \leq -3$, there is no possibility compatible with $-5/2 \leq L_n^* \leq 2$. This is an impossible case.
- If $-5/2 \leq \widehat{L}_n^* \leq -1$, then the only possibility is $L_n^* = \widehat{L}_n^*$. Therefore $d_n = -1$ is a correct choice.
- If $-1/2 \leq \widehat{L}_n^* \leq 0$, then the only possibility is $L_n^* = \widehat{L}_n^*$. Therefore $d_n = 0$ is a correct choice.
- If $1/2 \leq \widehat{L}_n^* \leq 2$, then the only possibility is $L_n^* = \widehat{L}_n^*$, and $d_n = 1$ is a correct choice.
- If $2+1/2 \leq \widehat{L}_n^* \leq 5$, there is no possibility compatible with $-5/2 \leq L_n^* \leq 2$. This is an impossible case.
- If $5 + 1/2 \leq \widehat{L}_n^* \leq 7$, then the only possibility is $L_n^* = \widehat{L}_n^* - 8$. This gives $-5/2 \leq L_n^* \leq -1$. Therefore $d_n = -1$ is a correct choice.
- If $\widehat{L}_n^* = 7 + 1/2$, then the only possibility is $L_n^* = \widehat{L}_n^* - 8$. This gives $L_n^* = -1/2$. Therefore $d_n = 0$ is a correct choice.

Therefore, the choice of d_n only needs the examination of four digits of L_n. This choice can be implemented by first converting $\widehat{L}_n^*$ to nonredundant representation (using a fast 4-bit adder), then by looking up in a 4-address bit table, or by directly looking up in a 8-address bit table, without preliminary addition. The digits of weight greater than or equal to 2^3 of $2^n L_n$ will never be used again; therefore there is no need to store them. This algorithm is very similar to an SRT division [179]. A slightly different solution, used by Takagi [443], consists of rewriting L_n so that the digit $L_{n,2}^*$ becomes null. This is always possible since $|2^n L_n|$ is less than $5/2$. After this, we only need to examine three digits of L_n at each step, instead of four. Takagi's solution is explained with more details in Section 9.6.

Carry-Save implementation

Assume now that we use the carry-save system. Define L_n^* as $2^n L_n$ *truncated after the first fractional digit*. We have

$$0 \leq 2^n L_n - L_n^* \leq 1;$$

therefore, if we choose

$$d_n = \begin{cases} -1 \text{ if } & L_n^* \leq -3/2 \\ 0 \text{ if } -1 \leq & L_n^* \leq -1/2 \\ 1 \text{ if } & L_n^* \geq 0, \end{cases} \tag{8.21}$$

then L_{n+1} will be between s_{n+1} and r_{n+1}. The proof is very similar to the proof of the signed-digit algorithm.

As in the previous section, we define $\widehat{L}_n^*$ as the 4-digit number obtained by truncating, in the carry-save representation of L_n^*, the digits of a weight greater than or equal to 2^3; that is, if

$$L_n^* = \cdots L_{n,4}^* L_{n,3}^* L_{n,2}^* L_{n,1}^* L_{n,0}^* . L_{n,-1}^*$$

with $L_{n,3}^* = 0, 1, 2$, then

$$\widehat{L}_n^* = L_{n,2}^* L_{n,1}^* L_{n,0}^* . L_{n,-1}^*.$$

We have

- $0 \leq \widehat{L}_n^* \leq 15$,
- $L_n^* - \widehat{L}_n^*$ is a multiple of 8.

Moreover, from $-5/2 < 2^n L_n < 2$ and $0 \leq 2^n L_n - L_n^* \leq 1$, we get $-7/2 < L_n^* < 2$ and, since L_n^* is a multiple of $1/2$, this gives $-3 \leq L_n^* \leq 3/2$. Therefore:

- If $0 \leq \widehat{L}_n^* \leq 3/2$, then the only possibility compatible with $-3 \leq L_n^* \leq 3/2$ is $L_n^* = \widehat{L}_n^*$; therefore $d_n = 1$ is a correct choice.
- If $2 \leq \widehat{L}_n^* \leq 4 + 1/2$, there is no possibility compatible with $-3 \leq L_n^* \leq 3/2$. This is an impossible case.
- If $5 \leq \widehat{L}_n^* \leq 6 + 1/2$, then $L_n^* = \widehat{L}_n^* + 8$ and $d_n = -1$ is a correct choice.
- If $7 \leq \widehat{L}_n^* \leq 7 + 1/2$, then $L_n^* = \widehat{L}_n^* + 8$ and $d_n = 0$ is a correct choice.
- If $8 \leq \widehat{L}_n^* \leq 9 + 1/2$, then $L_n^* = \widehat{L}_n^* + 8$ and $d_n = 1$ is a correct choice.
- If $10 \leq \widehat{L}_n^* \leq 12 + 1/2$, there is no possibility compatible with $-3 \leq L_n^* \leq 3/2$. This is an impossible case.
- If $13 \leq \widehat{L}_n^* \leq 14 + 1/2$, then $L_n^* = \widehat{L}_n^* + 16$ and $d_n = -1$ is a correct choice.
- If $\widehat{L}_n^* = 15$, then $\widehat{L}_n^* = -1$ and $d_n = 0$ is a correct choice.

Therefore the choice of d_n only needs the examination of four digits of L_n. As for the signed-digit version of the algorithm, this choice can be implemented by first converting $\widehat{L}_n^*$ to nonredundant representation (using a fast 4-digit adder), then by looking up in a 4-address bit table.

8.3.2 Faster Computation of Logarithms

Now assume that we want to compute logarithms quickly. Some notations adopted here are taken from [357], and Asger-Munk Nielsen helped to perform this study. We start from (8.17), that is, from the basic iteration:

$$\begin{cases} t_{n+1} &= t_n + \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n}, \end{cases}$$

with $n \geq 1$, and slightly modify it as follows,

$$\begin{cases} L_{n+1} &= L_n - \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n}, \end{cases} \tag{8.22}$$

where, as in the previous section, $L_n = t - t_n$. Since $E_n \times \exp(L_n)$ remains constant, if we are able to find a sequence of terms $d_k \in \{-1, 0, 1\}$ such that E_n goes to 1, then we will have $L_n \to L_1 + \ln(E_1)$.

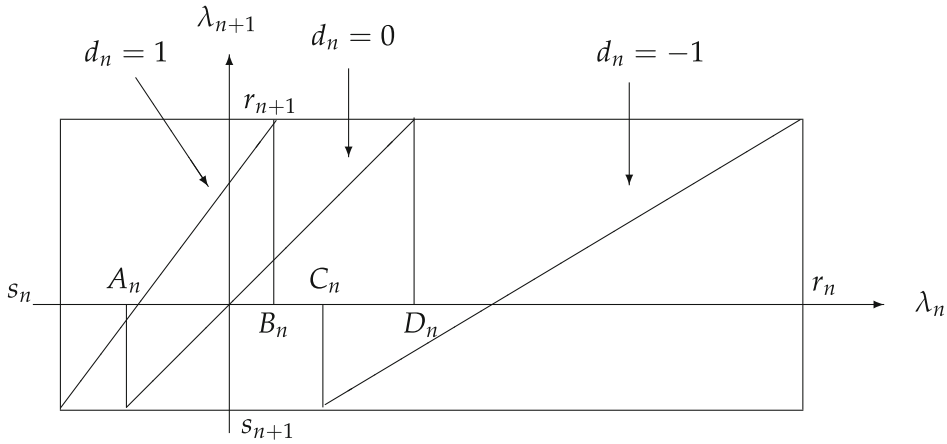


Figure 8.7 *Robertson diagram for the logarithm. The three straight lines give* $\lambda_{n+1} = \lambda_n(1 + d_n 2^{-n}) + d_n 2^{-n}$ *for* $d_n = -1, 0, 1$.

Define $\lambda_n = E_n - 1$. To compute logarithms, we want λ_n to go to zero as k goes to infinity. The Robertson diagram in Figure 8.7 displays the value of λ_{n+1} versus λ_n (i.e., $\lambda_{n+1} = \lambda_n(1 + d_n 2^{-n}) + d_n 2^{-n}$), for all possible values of d_n. In this diagram, r_n satisfies $r_{n+1} = \left(1 - 2^{-n}\right) r_n - 2^{-n}$, and s_n satisfies $s_{n+1} = \left(1 + 2^{-n}\right) s_n + 2^{-n}$. This gives:

$$r_n = \sum_{k=n}^{\infty} 2^{-k} \prod_{j=n}^{k} \frac{1}{1 - 2^{-j}}$$
$$s_n = -\sum_{k=n}^{\infty} 2^{-k} \prod_{j=n}^{k} \frac{1}{1 + 2^{-j}}.$$

One can show that r_n and s_n go to 0 as n goes to $+\infty$. According to this diagram (and assuming $A_n \le B_n \le C_n \le D_n$), any choice of d_n that satisfies (8.23) will ensure $\lambda_n \in [s_n, r_n]$, which implies $\lambda_n \to 0$.

$$\begin{cases} \text{if} \quad \lambda_n < A_n & \text{then } d_n = 1 \\ \text{if } A_n \le \lambda_n \le B_n & \text{then } d_n = 1 \text{ or } 0 \\ \text{if } B_n < \lambda_n < C_n & \text{then } d_n = 0 \\ \text{if } C_n \le \lambda_n \le D_n & \text{then } d_n = 0 \text{ or } -1 \\ \text{if} \quad D_n < \lambda_n & \text{then } d_n = -1. \end{cases} \tag{8.23}$$

The values A_n, B_n, C_n, and D_n satisfy

$$\begin{aligned} A_n &= s_{n+1} \\ B_n &= \frac{r_{n+1} - 2^{-n}}{1 + 2^{-n}} \\ C_n &= \frac{s_{n+1} + 2^{-n}}{1 - 2^{-n}} \\ D_n &= r_{n+1}. \end{aligned}$$

It follows, using these relations, that $A_n < B_n < C_n < D_n$ for all $n \ge 1$. Table 8.3 gives the first values and limits of 2^n times these values.

Table 8.3 *First 5 values and limit values of* $2^n s_n$, $2^n r_n$, $2^n A_n$, $2^n B_n$, $2^n C_n$, and $2^n D_n$.

n	$2^n s_n$	$2^n r_n$	$2^n A_n$	$2^n B_n$	$2^n C_n$	$2^n D_n$
1	−1.161	4.925	−0.74	0.30	0.51	1.46
2	−1.483	2.925	−0.85	0.15	0.19	1.19
3	−1.709	2.388	−0.92	0.08	0.09	1.09
4	−1.844	2.179	−0.96	0.04	0.04	1.04
5	−1.920	2.086	−0.98	0.02	0.02	1.02
∞	−2	2	−1	0	0	1

One can show that for any $n \geq 1$:

$$\begin{aligned} 2^n s_n &\geq -2 \\ 2^n r_n &\leq 5 \\ 2^n A_n &\leq -1/2 \\ 2^n B_n &\geq 0 \\ 2^n C_n &\leq 1 \\ 2^n D_n &\geq 1. \end{aligned}$$

Moreover, if $n \geq 2$, then $2^n C_n \leq 1/2$ and $2^n r_n \leq 3$. Let us see how the choice of d_n can be carried out using signed-digit arithmetic.

Signed-digit implementation

Assume that we use the radix-2 signed-digit system. Define λ_n^* as $2^n \lambda_n$ *truncated after the first fractional digit*. We have

$$\left|\lambda_n^* - 2^n \lambda_n\right| \leq \frac{1}{2}.$$

Therefore, *if n is greater than or equal to* 2, we can choose

$$d_n = \begin{cases} +1 & \text{if } \lambda_n^* \leq -1/2 \\ 0 & \text{if } \lambda_n^* = 0 \text{ or } 1/2 \\ -1 & \text{if } \lambda_n^* \geq 1. \end{cases} \tag{8.24}$$

If $n = 1$, then we need two fractional digits of the signed-digit representation of λ_1. And yet, in many cases, λ_1 will be in conventional (i.e., nonredundant, with digits 0 or 1) binary representation (in practice, if we want to compute the logarithm of a floating-point number x, λ_1 is obtained by suppressing the leading "1" of the mantissa of x; incidentally, this "1" may not be stored if the floating-point format uses the "hidden bit" convention — see Section 2.1.1). Knowing this, if $\lambda_1^* = 2\lambda_1$ truncated after the *first* fractional bit, then

$$0 \leq \lambda_1 - \lambda_1^* \leq 1/2$$

and we can choose

$$d_1 = \begin{cases} 0 & \text{if } \lambda_1^* = 0, 1/2 \\ -1 & \text{if } \lambda_1^* \geq 1. \end{cases}$$

Therefore (8.24) can be used for all n, provided that λ_1 is represented in the nonredundant binary number system. The convergence domain of the algorithm is

$$L_1 \in [s_1 + 1, r_1 + 1] = [0.4194\cdots, 3.4627\cdots].$$

Table 8.4 *The first digits of the first 15 values* $w_i = \ln(1 + 2^{-i})$. *As* i *increases,* w_i *gets closer to* 2^{-i}.

i	$\ln(1 + 2^{-i})$
0	$0.10110001011100100001011111101111101000111\cdots$
1	$0.01100111110011001000111110110010111111001\cdots$
2	$0.00111001000111111110111110001111001101010 0\cdots$
3	$0.00011110001001110000011101101110001010101 1\cdots$
4	$0.00001111100001010001100001100000000100010\cdots$
5	$0.00000111111000001010011011000011100111100 0\cdots$
6	$0.00000011111110000001010100010110000111111 0\cdots$
7	$0.00000001111111100000001010100110101100010 0\cdots$
8	$0.00000000111111111000000001010101000101011 0\cdots$
9	$0.00000000011111111110000000001010101001101 0\cdots$
10	$0.00000000001111111111000000000010101010010 0\cdots$
11	$0.00000000000111111111110000000000010101010 1\cdots$
12	$0.00000000000011111111111100000000000010101\cdots$
13	$0.00000000000001111111111111000000000000010\cdots$
14	$0.00000000000000111111111111110000000000000\cdots$
15	$0.00000000000000011111111111111100000000000\cdots$

8.4 Baker's Predictive Algorithm

If i is large enough, then $\ln(1+2^{-i})$ and $\arctan 2^{-i}$ are very close to 2^{-i} (this can be seen by examining Table 8.4). Baker's *predictive algorithm* [29], originally designed for computing the trigonometric functions[6] but easily generalizable to exponentials and logarithms, is based on this remark. We have already seen how the sequence $\ln(1+2^{-i})$ can be used for computing functions. The sequence $\arctan 2^{-i}$ is used by the CORDIC algorithm (see next chapter).

From the power series

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \cdots$$

one can easily show that

$$0 < 2^{-i} - \ln(1+2^{-i}) < 2^{-2i-1}. \tag{8.25}$$

Similarly, one can show that

$$0 < 2^{-i} - \arctan(2^{-i}) < \frac{1}{3}2^{-3i}. \tag{8.26}$$

This means that if x is small enough, the first terms of the decomposition of x on the discrete base $\ln(1+2^{-i})$ or $\arctan 2^{-i}$ are likely to be the same as the first terms of its decomposition on the base (2^{-i}), that is, its binary decomposition. This is illustrated by Table 8.5.

[6] Using a modified CORDIC algorithm; CORDIC is presented in the next chapter.

Table 8.5 *Comparison among the binary representations and the decompositions (given by the restoring algorithm) on the discrete bases* $\ln(1+2^{-i})$ *and* $\arctan 2^{-i}$ *for some values of* x*. When* x *is very small the different decompositions have many common terms.*

x	w_i	decomposition
$\frac{1}{2}$	2^{-i} (binary)	01000000000000000000
	$\ln(1+2^{-i})$	01001100011001101101
	$\arctan 2^{-i}$	01000100101001111000
$\frac{1}{10}$	2^{-i} (binary)	00001100110011001001
	$\ln(1+2^{-i})$	00001101000110101101
	$\arctan 2^{-i}$	00001100110011111001
$\frac{1}{1024}$	2^{-i} (binary)	00000000001000000000000000000000000000
	$\ln(1+2^{-i})$	00000000001000000000001111111111010101
	$\arctan 2^{-i}$	00000000001000000000000000000000010101

Let w_i be $\ln(1+2^{-i})$ or $\arctan(2^{-i})$ depending on the function we wish to compute. Assume that we are at some step $n \geq 2$ of a decomposition method. We have a value x_n that satisfies:

$$0 \leq x_n \leq \sum_{k=n}^{\infty} w_k < \sum_{k=n}^{\infty} 2^{-k} = 2^{-n+1}$$

and we need to find values $d_n, d_{n+1}, d_{n+2} \ldots$, such that

$$x_n = \sum_{k=n}^{\infty} d_k w_k.$$

Let

$$0.00000 \cdots 0 d_n^{(n)} d_{n+1}^{(n)} d_{n+2}^{(n)} d_{n+3}^{(n)} \cdots$$

be the binary representation of x_n, that is,

$$x_n = \sum_{k=n}^{\infty} d_k^{(n)} 2^{-k}.$$

Since w_p is very close to 2^{-p}, the basic idea is to choose:

$$\begin{cases} d_n & = d_n^{(n)} \\ d_{n+1} & = d_{n+1}^{(n)} \\ & \vdots \\ d_\ell & = d_\ell^{(n)}, \end{cases} \tag{8.27}$$

for some ℓ. This gives values (d_i) without having to perform any comparison or table lookup. Of course, since the sequences (w_p) and (2^{-p}) are not exactly equal, this process will not always give a correct result without modifications. A *correction step* is necessary. Define

$$\tilde{x}_{\ell+1} = x_n - d_n w_n - d_{n+1} w_{n+1} - \cdots - d_\ell w_\ell.$$

We have:

$$\tilde{x}_{\ell+1} = x_n - \sum_{k=n}^{\ell} d_k w_k = x_n - \sum_{k=n}^{\ell} d_k 2^{-k} + \sum_{k=n}^{\ell} d_k (2^{-k} - w_k).$$

Therefore

$$0 \le \tilde{x}_{\ell+1} \le 2^{-\ell} + \sum_{k=n}^{\ell} (2^{-k} - w_k).$$

- If $w_i = \ln\left(1 + 2^{-i}\right)$, using (8.25), this gives

$$0 \le \tilde{x}_{\ell+1} < 2^{-\ell} + \frac{2^{-2n+1}}{3}, \tag{8.28}$$

- and if $w_i = \arctan 2^{-i}$, using (8.26), this gives

$$0 \le \tilde{x}_{\ell+1} < 2^{-\ell} + \frac{2^{-3n+3}}{21}. \tag{8.29}$$

In both cases, $\tilde{x}_{\ell+1}$ is small. Now let us find a convenient value for ℓ.

- If $w_i = \ln\left(1 + 2^{-i}\right)$, let us choose $\ell = 2n - 1$. This gives

$$0 \le \tilde{x}_{2n} \le 2^{-2n+1}\left(1 + \frac{1}{3}\right).$$

  The "correction step" consists of again using the constant $w_\ell = w_{2n-1}$ in the decomposition.[7] Define δ_ℓ as 1 if $\tilde{x}_{\ell+1} > w_l$, else 0, and $x_{2n} = x_{\ell+1} = \tilde{x}_{\ell+1} - \delta_\ell w_l$. We get:

  – if $\delta_\ell = 0$, then $0 \le x_{\ell+1} \le w_\ell \le \sum_{k=l+1}^{\infty} w_k$;
  – if $\delta_\ell = 1$, then from (8.28) and the Taylor expansion of the logarithm, we get:

$$\begin{aligned} 0 \le x_{\ell+1} = \tilde{x}_{\ell+1} - w_l &\le 2^{-2n+1}\left(1 + \frac{1}{3}\right) - 2^{-2n+1} \\ &\quad + \frac{1}{2} 2^{-4n+2} \\ &\le 2^{-2n}\left(\frac{2}{3} + 2^{-2n+1}\right) \le 2^{-2n} \\ &\quad \text{(since } n \ge 2\text{)} \\ &\le \sum_{k=l+1}^{\infty} w_k. \end{aligned}$$

[7] This can be viewed as the possible use of $d_i = 2$ for a few values of i, or as the use of a new discrete base, obtained by repeating a few terms of the sequence (w_i).

- If $w_i = \arctan 2^{-i}$, let us choose $\ell = 3n - 1$. This gives

$$0 \leq \tilde{x}_{3n} \leq 2^{-3n+1}\left(1 + \frac{1}{3}\right).$$

The "correction step" consists of using the constant $w_\ell = w_{3n-1}$ again in the decomposition. Define δ_ℓ as 1 if $\tilde{x}_{\ell+1} > w_\ell$, else 0, and $x_{\ell+1} = x_{3n} = \tilde{x}_{\ell+1} - \delta_\ell w_\ell$. We get

– if $\delta_\ell = 0$, then

$$0 \leq x_{\ell+1} \leq w_\ell \leq \sum_{k=\ell+1}^{\infty} w_k;$$

– if $\delta_\ell = 1$, then, from (8.29) and the Taylor expansion of the arctangent function, we get:

$$\begin{aligned} 0 \leq x_{\ell+1} = \tilde{x}_{\ell+1} - w_\ell &\leq \frac{25}{21}2^{-3n+1} - 2^{-3n+1} + \frac{2^{-9n+3}}{3} \\ &\leq 2^{-3n}\left(\frac{8}{21} + \frac{8}{3}2^{-6n}\right) \\ &\leq 2^{-3n}(\text{ since } n \geq 2) \\ &\leq \sum_{k=\ell+1}^{\infty} w_k. \end{aligned}$$

Therefore, in both cases, we have got a new value n' ($n' = \ell + 1$) that satisfies:

$$0 \leq x_{n'} \leq \sum_{k=n'}^{\infty} w_k,$$

and we can start the estimation of the next terms of the decomposition from the binary expansion of $x_{n'}$.

Assume that we use the sequence $w_n = \ln\left(1 + 2^{-n}\right)$. We cannot start the method from $n = 0$ or $n = 1$ ($n = 0$ would give $\ell = -1$, and $n = 1$ would give $\ell = 1$). Starting from a small value of n larger than 1 would not allow us to predict many values d_i: $n = 3$ would give $\ell = 5$. This would allow us to find d_3, d_4, and d_5, but we would have to perform a "correction step" immediately afterwards. To make Baker's method efficient, we must handle the first values of n using a different algorithm. A solution is to use a small table. Let us examine how this can be done.

For an m-bit chain $(\alpha_0, \alpha_1, \ldots, \alpha_{m-1})$ define $I_{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}}$ as the interval containing all the real numbers whose binary representation starts with $\alpha_0.\alpha_1\alpha_2\cdots\alpha_{m-1}$, that is,

$$I_{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}} = \left[\alpha_0.\alpha_1\alpha_2\cdots\alpha_{m-1}, \alpha_0.\alpha_1\alpha_2\cdots\alpha_{m-1} + 2^{-m-1}\right].$$

For an n-bit chain $(d_0, d_1, \ldots, d_{n-1})$, define $J_{d_0,d_1,\ldots,d_{n-1}}$ as the interval

$$J_{d_0,\ldots,d_{n-1}} = \left[d_0w_0 + \cdots + d_{n-1}w_{n-1}, d_0w_0 + \cdots + d_{n-1}w_{n-1} + \sum_{i=n}^{\infty} w_i\right].$$

The interval $J_{d_0,d_1,\ldots,d_{n-1}}$ contains the numbers t that can be written

$$t = \sum_{k=0}^{\infty} \delta_k w_k,$$

where $\delta_k = d_k$ for $k \leq n-1$ and $\delta_k = 0, 1$ for $k \geq n$.

We can start Baker's algorithm at step n if there exists a number m such that for every possible m-bit chain $(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1})$ such that $x_0 = \alpha_0.\alpha_1\alpha_2\cdots\alpha_{m-1}\cdots$ belongs to the convergence domain of the restoring algorithm,[8] there exists an n-bit chain $(d_0, d_1, \ldots, d_{n-1})$ such that $J_{d_0,d_1,\ldots,d_{n-1}}$ contains $I_{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}}$.

If this is true, once we have computed $d_0, \ldots, d_{n-1}$ for every possible value of $\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{m-1}$, it suffices to store the values $d_0, d_1, \ldots, d_{n-1}$ in an m-address bit table. Unfortunately, m turns out to be too large for convenient values of n: After some experiments, m seems to be larger than $2n$ (for $2 \leq n \leq 8$, and $w_k = \ln\left(1 + 2^{-k}\right)$, m is equal to $2n + 1$). A solution is to perform a correction step after the initial table lookup. Instead of providing, for each m-bit chain $(\alpha_0\alpha_1\alpha_2\cdots\alpha_{m-1})$, an n-bit chain such that $J_{d_0,d_1,\ldots,d_{n-1}}$ contains $I_{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}}$, we give an n-bit chain such that $J'_{d_0,d_1,\ldots,d_{n-1}}$ contains $I_{\alpha_0,\alpha_1,\ldots,\alpha_{m-1}}$, where $J'_{d_0,d_1,\ldots,d_{n-1}}$ is the interval

$$\left[d_0w_0 + \cdots + d_{n-1}w_{n-1}, d_0w_0 + \cdots + d_{n-1}w_{n-1} + w_{n-1} + \sum_{i=n}^{\infty} w_i\right].$$

This requires smaller values of m. For $2 \leq n \leq 10$, $m = n + 1$ suffices. Once $d_0, d_1, \ldots, d_{n-1}$ is obtained from the table, it suffices to compute

$$x_n^{(0)} = x_0 - d_0w_0 - d_1w_1 - \cdots - d_{n-1}w_{n-1}$$

and

$$x_n^{(1)} = x_0 - d_0w_0 - d_1w_1 - \cdots - d_{n-1}w_{n-1} - w_{n-1}$$

using redundant additions and a final conversion to nonredundant representation. Then we can start Baker's algorithm from

$$x_n = \begin{cases} x_n^{(0)} & \text{if } \quad x_n^{(1)} < 0 \\ x_n^{(1)} & \text{otherwise.} \end{cases}$$

The following Maple program computes the value of m and builds the table from any value of n, for $w_k = \ln\left(1 + 2^{-k}\right)$.

```
find_Baker := proc(n)
# finds the smallest number m of digits of the input number
# that allows us to start Baker's algorithm at step n
# and builds the table
# we start with m = n+1, if we succeed, we build the table
# else we increment m
  global m, TAB, failure;
  Digits := 30;
```

[8] That is, $0 \leq x_0 \leq \sum_{k=0}^{\infty} w_k$.

```
# recalculation of the convergence domain [0,A]
    A := evalf(log(2));
    for i from 1 to 100 do
            A := A+evalf(log(1+2^(-i))) od;
# First, we build the J intervals
  remainder := evalf(log(1+2^(-n)));
  for i from (n+1) to 100 do
      remainder := remainder+evalf(log(1+2^(-i)))
  od;
  for counter from 0 to (2^n-1) do
# computation of d_0, d_1, ... d_n-1
# where the d_i's are the digits of the binary representation
# of counter
    decomp := counter;
    for k from 1 to n do
       d[n-k] := decomp mod 2;
       decomp := trunc(decomp/2)
    od;
    Jleft[counter] := evalf(d[0]*log(2));
    for i from 1 to (n-1) do
        Jleft[counter] := Jleft[counter]
                        + evalf(d[i]*log(1+2^(-i)))
    od;
    Jright[counter] := Jleft[counter]
        + remainder+evalf(log(1+2^(-n+1)));
  od;
# now we try successive values of m
  m := n;
  failure := true;
# failure = true means that m is not large enough
  while (failure) do
     m := m+1;
     powerof2 := 2^(m-1);
     Ileft[0] := 0; Iright[0] := evalf(1/powerof2);
     for counter from 1 to 2^m-1 do ;
        Ileft[counter] := Iright[counter-1];
        Iright[counter] :=  evalf((counter+1)/powerof2);
     od;
# Now we must check if for each I-interval included in the
# convergence domain of the algorithm
# there exists a J-interval that
# contains it
  countermax := trunc(A*2^(m-1))-1;
  Jstart := 0;
  failure := false;
  counter := 0;
  while (not failure) and (counter <= countermax) do
    while Jright[Jstart] < Iright[counter] do
```

```
            Jstart := Jstart+1
      od;
      if Jleft[Jstart] <= Ileft[counter] then
        TAB[counter] := Jstart else failure := true
      fi;
      counter := counter+1
    od;
    if (failure=false)
       then
         print (m);
         for counter from 0 to countermax do
             print(counter,TAB[counter])
         od
    fi
  od;
end;
```

Now let us examine an example.

Example 10 (Computation of the exponential function). *We want to compute the exponential of* $x = 0.110010110_2 = 0.79296875_{10}$ *using Baker's method, and we use Table 8.6, which was built using the previously presented method, with* $n = 4$ *and* $m = 5$. *In our example, the table gives* $d_0 = 0$, $d_1 = 1$, $d_2 = 1$, *and* $d_3 = 0$. *So we compute*

$$x_4^{(0)} = x_0 - d_0 w_0 - d_1 w_1 - d_2 w_2 - d_3 w_3$$

and

$$x_4^{(1)} = x_0 - d_0 w_0 - d_1 w_1 - d_2 w_2 - d_3 w_3 - w_3;$$

this gives

$$\begin{cases} x_4^{(0)} = 0.16436009\cdots_{10} \\ x_4^{(1)} = 0.04657705\cdots_{10}. \end{cases}$$

Since $x_4^{(1)} \geq 0$, *we start Baker's method from* $x_4 = x_4^{(1)}$. *Since* $n = 4$, $\ell = 7$ *so we can deduce* d_4, d_5, d_6, *and* d_7 *from the binary representation of* x_4, *that is,* $0.000010111111011000111 10\cdots_2$. *This gives*

$$d_4 = 0, d_5 = 1, d_6 = 0, d_7 = 1;$$

therefore

$$\begin{aligned} \tilde{x}_8 &= x_4 - d_4 w_4 - d_5 w_5 - d_6 w_6 - d_7 w_7 \\ &= 0.0080232558\cdots_{10} \\ &= 0.0000001000001101110\cdots_2. \end{aligned}$$

Table 8.6 *Table obtained for $n = 4$ using our Maple program.*

First 5 Bits of x	First Terms d_i
00000	0000
00001	0000
00010	0000
00011	0001
00100	0001
00101	0010
00110	0010
00111	0011
01000	0011
01001	0100
01010	0101
01011	0101
01100	0110
01101	0111
01110	0111
01111	1001
10000	1010
10001	1010
10010	1011
10011	1011
10100	1100
10101	1101
10110	1101
10111	1110

Now we have to perform a correction step. Let us subtract w_7 from $\tilde{x}_8$. This gives $0.0002411153\cdots_{10}$, *which is positive. Therefore $\delta_7 = 1$ and*

$$\begin{aligned} x_8 &= \tilde{x}_8 - \delta_7 w_7 \\ &= 0.0002411153\cdots_{10} \\ &= 0.00000000000011111100110100111110101\cdots_2 . \end{aligned}$$

From the binary representation of x_8, we can deduce d_8, d_9, ..., d_{15}. This gives

$$d_8 = d_9 = d_{10} = d_{11} = d_{12} = 0, d_{13} = d_{14} = d_{15} = 1.$$

Therefore, taking into account the correction steps, we find

$$\begin{aligned} x = w_1 + w_2 + w_3 \quad & tabulation\ and\ correction \\ +w_5 + w_7 + w_7 \quad & 1st\ step\ of\ Baker's\ method \\ +w_{13} + w_{14} + w_{15} + \ldots \quad & 2nd\ step\ of\ Baker's\ method. \end{aligned}$$

This gives

$$\begin{aligned} e^x &= \left(1+\frac{1}{2^{-1}}\right)\left(1+\frac{1}{2^{-2}}\right)\left(1+\frac{1}{2^{-3}}\right) \\ &\quad \left(1+\frac{1}{2^{-5}}\right)\left(1+\frac{1}{2^{-7}}\right)\left(1+\frac{1}{2^{-7}}\right) \\ &\quad \left(1+\frac{1}{2^{-13}}\right)\left(1+\frac{1}{2^{-14}}\right)\left(1+\frac{1}{2^{-15}}\right) \\ &\approx 2.2099. \end{aligned}$$

8.5 Bibliographic Notes

The CORDIC algorithm (see Chapter 9) is a shift-and-add algorithm that allows evaluation of trigonometric and hyperbolic functions. It was introduced by Volder in 1959 [465]. Meggitt [337] presented the same basic iterations slightly differently, and saw them as "pseudomultiplication" and "pseudodivision" processes. The basic iterations for computing logarithms and exponentials (as well as iterations similar to CORDIC for the elementary functions) were presented by Specker [429] in 1965. Similar algorithms were studied by Linhardt and Miller [321]. An analysis of shift-and-add algorithms for computing the elementary functions was given by DeLugish [151] in 1970.

The CORDIC Algorithm

9

9.1 Introduction

The CORDIC algorithm was introduced in 1959 by Volder [465, 466]. In Volder's version, CORDIC makes it possible to perform rotations (and therefore to compute sine, cosine, and arctangent functions) and to multiply or divide numbers using only shift-and-add elementary steps. To quote Volder [466], the CORDIC technique was born out of necessity, the incentive being the replacement of the analog navigation computer of the B-58 bomber aircraft by a digital computer. The main challenge was the real-time determination of present position on a spherical earth.

The Hewlett-Packard 9100 desktop calculator, built in 1968[1], used CORDIC for the trigonometric functions.

In 1971, Walther [470, 471] generalized this algorithm to compute logarithms, exponentials, and square roots. CORDIC is not the fastest way to perform multiplications or to compute logarithms and exponentials but, since the same algorithm allows the computation of most mathematical functions using very simple basic operations, it is attractive for hardware implementations. CORDIC has been implemented in many pocket calculators since Hewlett-Packard's HP 35 [92], and in arithmetic processors or coprocessors such as the Intel 8087 [359] and some of its followers. Some authors have proposed the use of CORDIC processors for QR decomposition [318, 323], for signal processing applications (DFT [33, 154, 200, 482], discrete Hartley transform [77], filtering [152], SVD [74, 75, 178, 239, 323, 294]), for computer graphics [293], or for solving linear systems [4]. In 2009, for the 50th birthday of CORDIC, Meher et al. gave a very interesting survey [338].

9.2 The Conventional CORDIC Iteration

Volder's algorithm is based upon the following iteration,

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n}. \end{cases} \tag{9.1}$$

[1] See http://www.decodesystems.com/hp9100.html.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_9

The terms $\arctan 2^{-n}$ are precomputed and stored, and the d_is are equal to -1 or $+1$. In the *rotation mode* of CORDIC, d_n is chosen equal to the sign of z_n (i.e., $+1$ if $z_n \geq 0$, else -1). If $|z_0|$ is less than or equal to

$$\sum_{k=0}^{\infty} \arctan 2^{-k} = 1.7432866204723400035\cdots,$$

then

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K \times \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix}, \tag{9.2}$$

where the *scale factor* K is equal to $\prod_{n=0}^{\infty} \sqrt{1+2^{-2n}} = 1.646760258121\cdots$. For instance, to compute the sine and the cosine of a number θ, with $|\theta| \leq \theta_{\max} = \sum_{k=0}^{\infty} \arctan 2^{-k}$, we choose

$$\begin{aligned} x_0 &= 1/K = 0.6072529350088812561694\cdots \\ y_0 &= 0 \\ z_0 &= \theta. \end{aligned}$$

The bound $\sum_{k=0}^{\infty} \arctan 2^{-k} = 1.7432866204723400035\cdots$ is rather comfortable for evaluating trigonometric functions, since it is larger than $\pi/2$.

Now let us show how CORDIC works. That algorithm is based on the decomposition of $\theta = z_0$ on the discrete base (see Chapter 8) $w_n = \arctan 2^{-n}$, using the nonrestoring algorithm (see Theorem 17). The nonrestoring algorithm gives a decomposition of θ

$$\theta = \sum_{k=0}^{\infty} d_k w_k, \quad d_k = \pm 1, \quad w_k = \arctan 2^{-k}.$$

The basic idea of the *rotation mode* of CORDIC is to perform a rotation of angle θ as a sequence of elementary rotations of angles $d_n w_n$. We start from (x_0, y_0), and obtain the point (x_{n+1}, y_{n+1}) from the point (x_n, y_n) by a rotation of angle $d_n w_n$. This gives:

nonrestoring decomposition

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } t_n \leq \theta \\ -1 & \text{otherwise;} \end{cases} \end{aligned} \tag{9.3}$$

***n*th rotation**

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} \cos(d_n w_n) & -\sin(d_n w_n) \\ \sin(d_n w_n) & \cos(d_n w_n) \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}. \tag{9.4}$$

This can be simplified, first by noticing that, since $d_n = \pm 1$, $\cos(d_n w_n) = \cos(w_n)$ and $\sin(d_n w_n) = d_n \sin(w_n)$, then using the relation $\tan w_n = 2^{-n}$. We then replace (9.4) by

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \cos(w_n) \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}. \tag{9.5}$$

In (9.5), the multiplication by $\cos(w_n) = 1/\sqrt{1+2^{-2n}}$ is the only "true" multiplication, since in radix 2 a multiplication by 2^{-n} reduces to a shift. To avoid this multiplication, instead of (9.5), we perform

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}, \tag{9.6}$$

which is the basic CORDIC step, in the *trigonometric* type of iteration. It is no longer a *rotation* of angle w_n, but a *similarity*, or a "rotation-extension" (i.e., the combination of a rotation and a multiplication by a real factor) of angle w_n and factor $1/\cos w_n$. The choice of d_n given by (9.3) can be slightly simplified. If we define $z_n = \theta - t_n$, we get

$$\begin{aligned} z_0 &= \theta \\ z_{n+1} &= z_n - d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } z_n \geq 0 \\ -1 & \text{otherwise.} \end{cases} \end{aligned} \tag{9.7}$$

To sum up, the sequence (x_n, y_n) defined by Eq. (9.6) and (9.7) will not converge to the rotation of angle θ of (x_0, y_0) but to the result of a similarity of angle θ, whose factor is the product K of all the elementary factors, applied to (x_0, y_0). This gives (9.2). Figure 9.1 presents one step of the algorithm.

Now let us focus on the *vectoring mode* of CORDIC. This mode is used for computing arctangents. Assume that we wish to evaluate $\theta = \arctan y_0/x_0$. The following algorithm converges provided that θ belongs to the convergence domain of the rotation mode (i.e., $|\theta| \leq \sum_{i=0}^{\infty} \arctan 2^{-i}$). To simplify, we assume here that both x_0 and y_0 are positive. First, imagine that *we already know* θ (this is a reasoning similar to the one used for deducing the restoring algorithm for logarithms from the algorithm for exponentials in Section 8.2.2). If we start from (x_0, y_0) and perform a rotation[2] of angle $-\theta$, using the rotation mode, then we compute the sequence

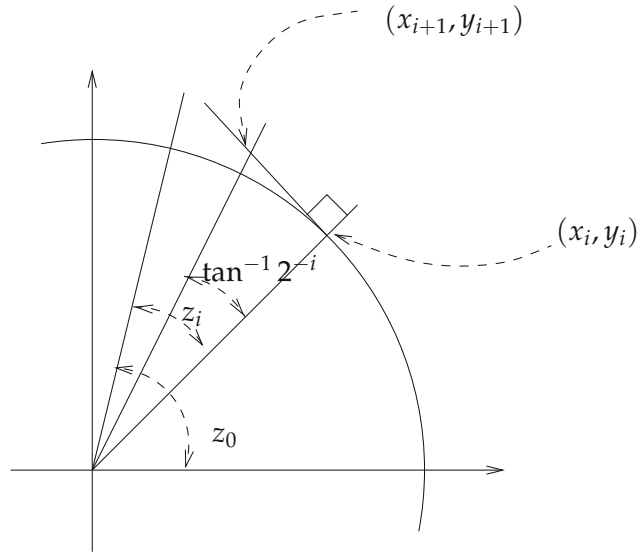


Figure 9.1 *One iteration of the CORDIC algorithm.*

[2] Or, more precisely, a similarity.

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n} \end{cases}$$

with $z_0 = -\theta$ and

$$d_n = \begin{cases} 1 & \text{if } z_n \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{9.8}$$

This gives

$$\begin{aligned} x_n &\to K\sqrt{x_0^2 + y_0^2} \\ y_n &\to 0 \\ z_n &\to 0. \end{aligned}$$

Now define a new variable z'_n equal to $\theta + z_n$. Since $z_n \geq 0 \Leftrightarrow z'_n \geq \theta$, we can perform the same iteration as previously and get the same results by choosing, instead of (9.8)

$$d_n = \begin{cases} 1 & \text{if } z'_n \geq \theta \\ -1 & \text{otherwise} \end{cases} \tag{9.9}$$

with $z'_0 = 0$. This gives $z'_n \to \theta$.

Now we have to take into account that θ is unknown: it is precisely the value we wish to compute! z'_n measures the opposite of the angle by which (x_0, y_0) must be rotated to get[3] (x_n, y_n). If we have rotated by an angle whose opposite is greater than θ, then (x_n, y_n) is *below* the x-axis; hence y_n is negative. Otherwise, y_n is positive. Therefore the test $z'_n \geq \theta$ can be replaced by $y_n \leq 0$.

By doing this, we no longer need to know θ, and we get the *vectoring mode* of CORDIC. In that mode, d_n is chosen equal to the sign of $(-y_n)$ (i.e., $+1$ if $y_n \leq 0$, else -1). This gives

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} K\sqrt{x_0^2 + y_0^2} \\ 0 \\ z_0 + \arctan \dfrac{y_0}{x_0} \end{pmatrix}, \tag{9.10}$$

where the constant K is the same as in the rotation mode.

Since trigonometric and hyperbolic functions are closely related, one may expect that a slight modification of Volder's algorithm could be used for the computation of hyperbolic functions. In 1971, John Walther [470] found the correct modification, and obtained the generalized CORDIC iteration:

$$\begin{cases} x_{n+1} = x_n - m d_n y_n 2^{-\sigma(n)} \\ y_{n+1} = y_n + d_n x_n 2^{-\sigma(n)} \\ z_{n+1} = z_n - d_n w_{\sigma(n)}, \end{cases} \tag{9.11}$$

where the results and the values of d_n, m, w_n, and $\sigma(n)$ are presented in Tables 9.1 and 9.2.

In the *hyperbolic* type of iteration ($m = -1$), the terms $i = 4, 13, 40, \ldots, k, 3k+1, \ldots$ (i.e., the terms $i = (3^{j+1} - 1)/2$) of the sequence $\tanh^{-1} 2^{-i}$ are used twice (this is why we need to use the function σ). This is necessary since the sequence $\tanh^{-1} 2^{-n}$ does not satisfy the condition of

[3] Neglecting the scale factor.

Table 9.1 *Computability of different functions using CORDIC.*

Type	m	w_k	$d_n = \text{sign} z_n$ (Rotation Mode)	$d_n = -\text{sign} y_n$ (Vectoring Mode)
circular	1	$\arctan 2^{-k}$	$x_n \to K\,(x_0 \cos z_0 - y_0 \sin z_0)$ $y_n \to K\,(y_0 \cos z_0 + x_0 \sin z_0)$ $z_n \to 0$	$x_n \to K\sqrt{x_0^2 + y_0^2}$ $y_n \to 0$ $z_n \to z_0 + \arctan \frac{y_0}{x_0}$
linear	0	2^{-k}	$x_n \to x_0$ $y_n \to y_0 + x_0 z_0$ $z_n \to 0$	$x_n \to x_0$ $y_n \to 0$ $z_n \to z_0 + \frac{y_0}{x_0}$
hyperbolic	-1	$\tanh^{-1} 2^{-k}$	$x_n \to K'\,(x_1 \cosh z_1 + y_1 \sinh z_1)$ $y_n \to K'\,(y_1 \cosh z_1 + x_1 \sinh z_1)$ $z_n \to 0$	$x_n \to K'\sqrt{x_1^2 - y_1^2}$ $y_n \to 0$ $z_n \to z_1 + \tanh^{-1} \frac{y_1}{x_1}$

Table 9.2 *Values of* $\sigma(n)$ *in Eq.* (9.11) *and Table* 9.1.

Circular ($m = 1$)	$\sigma(n) = n$
Linear ($m = 0$)	$\sigma(n) = n$
Hyperbolic ($m = -1$)	$\sigma(n) = n - k$ where k is the largest integer such that $3^{k+1} + 2k - 1 \leq 2n$

Theorem 17. The sequence $\tanh^{-1} 2^{-\sigma(n)}$, obtained from the sequence $\tanh^{-1} 2^{-n}$ by repeating the terms $4, 13, 40, \ldots$ satisfies that condition. The new scaling factor

$$K' = \prod_{i=1}^{\infty} \sqrt{1 - 2^{-2\sigma(i)}}$$

equals $0.828159360960215627076198 32\cdots$. Therefore, to compute $\cosh\theta$ and $\sinh\theta$, one should choose

$$\begin{aligned} x_1 &= 1/K' = 1.207497067763072128877 72\cdots \\ y_1 &= 0 \\ z_1 &= \theta. \end{aligned}$$

In the rotation mode, the maximum value for $|\theta|$ is $1.1181730155265\cdots$.

In Walther's version, CORDIC makes it possible to compute many mathematical functions. For instance, e^x is obtained by adding $\cosh x$ and $\sinh x$, and $\ln x$ is obtained using the relation

$$\ln(x) = 2\tanh^{-1}\left(\frac{x-1}{x+1}\right)$$

whereas the square root of x is obtained as

$$\sqrt{x} = K'\sqrt{\left(x + \frac{1}{4K'^2}\right)^2 - \left(x - \frac{1}{4K'^2}\right)^2}.$$

9.3 Acceleration of the Last Iterations

Assume we perform CORDIC iterations of the "trigonometric type" (9.1), in the rotation mode. From the well-known series expansion

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots,$$

we easily deduce that if, starting from iteration number n_0, we replace in (9.1) the term "arctan 2^{-n} ", by "2^{-n}" (i.e., with the notation of (9.11), we take $w_n = 2^{-n}$ for $n \geq n_0$), the error committed in the decomposition of the initial angle $\theta = z_0$ is bounded by

$$\frac{1}{3} \cdot \sum_{i=n_0}^{\infty} 2^{-3i} = \frac{8}{21} \cdot 2^{-3n_0}.$$

That bound is less than 2^{-m-1} as soon as $n_0 > m/3 + (4 - \log_2(21))/3 \approx m/3 - 0.130$. This means that if we want that error to be less than 2^{-m-1} (say, if we plan to perform m CORDIC iterations), we can replace arctan 2^{-n} by 2^{-n} for $n \geq m/3$. This leads to several possible strategies:

- since performing CORDIC iterations in rotation mode with $w_n = 2^{-n}$ is equivalent to performing a multiplication (see Table 9.1), if a fast multiplier is available, we can replace the last $2n/3$ iterations by one multiplication. This was suggested by Ahmed [3];
- for $n \geq n_0 = m/3$, we can suppress the z-part of iteration (9.1), and choose d_n equal to the binary digit of weight 2^{-n} of z_{n_0} (this is clearly a variant of Baker's predictive algorithm presented in Chapter 8). To implement this choice, we can either "recode" z_{n_0} so that its digits are equal to -1 or 1 only, or we can stay with the original digits 0 and 1 of z_0, and to avoid variable scale factor problems (see Section 9.5.3), we can replace the x and y parts of iteration (9.1) by

$$\begin{cases} x_{n+1} = x_n(1 - 2^{-(2n+1)}) - d_n y_n 2^{-n} \\ y_{n+1} = y_n(1 - 2^{-(2n+1)}) + d_n x_n 2^{-n}. \end{cases} \tag{9.12}$$

  By doing that, since for n large enough[4] $\sin(2^{-n}) \approx 2^{-n}$ and $\cos(2^{-n}) \approx 1 - 2^{-2n-1}$, we perform "true" rotations instead of similarities: this is the *scaling free* CORDIC scheme introduced by Maharatna et al. [328] and later on enhanced, using Booth recoding, by Jaime et al. [252].

9.4 Scale Factor Compensation

As we have seen before, in the trigonometric type of iteration, we do not perform a rotation of the initial vector (x_0, y_0), but the combination of a rotation and a multiplication by the factor

$$K = \prod_{i=0}^{+\infty} \sqrt{1 + 2^{-2i}}.$$

[4]Which is the case here since we assume $n \geq m/3$.

As we have seen previously, if we just want to evaluate sines and cosines, this multiplication by a scale factor is not a problem. And yet, if we actually want to perform rotations, we have to compensate for this multiplication. Despain [154] and Haviland and Tuszinsky [228] have suggested finding values $\alpha_i = -1, 0, +1$, for $i = 0, 1, 2 \ldots$ such that

$$\prod_{i=0}^{+\infty}(1 + \alpha_i 2^{-i}) = \frac{1}{K}$$

and merging the CORDIC iterations with multiplications (that reduce to shifts and additions) by the terms $(1 + \alpha_i 2^{-i})$, that is, performing iterations of the form

$$\begin{aligned} x_{n+1} &= \left(x_n - d_n y_n 2^{-n}\right)\left(1 + \alpha_n 2^{-n}\right) \\ y_{n+1} &= \left(y_n + d_n x_n 2^{-n}\right)\left(1 + \alpha_n 2^{-n}\right). \end{aligned}$$

There are several possible solutions for the sequence (α_n). One of them is given in Table 9.3. The following Maple session shows how the terms α_i can be computed:

```
> Digits := 30;
    Digits := 30

> Ksquare := 1;
    Ksquare := 1

> for j from 0 to 60 do
>    Ksquare := Ksquare * (1.0 + 2^(-2*j)) od:
> Ksquare;
    2.71181934772695876069108846971

> K := sqrt(Ksquare);
    K := 1.64676025812106564836605122228

> A := K;
    A := 1.64676025812106564836605122228
```

Table 9.3 *First values α_i that can be used in Despain's scale factor compensation method.*

i	α_i
0	0
1	–1
2	1
3	–1
4	1
5	1
6	1
7	–1
8	1
9	1
10	–1

```
> for i from 1 to 60 do
> if A > 1 then alpha[i] := -1: A := A*(1-2^(-i))
> else alpha[i] := 1: A := A*(1+2^(-i)) fi od;
> for i from 1 to 4 do print(alpha[i]) od;

                          -1
                           1
                          -1
                           1
```

Other scale factor compensation methods have been suggested. For instance, Deprettere, Dewilde, and Udo [152] choose, instead of rotation angles of the form $\arctan 2^{-n}$, angles of the form $\arctan\left(2^{-n} \pm 2^{-m}\right)$, where the terms $\pm 2^{-m}$ are chosen so that the scale factor becomes 1 (or 2). A discussion on scale factor compensation methods can be found in [455].

9.5 CORDIC With Redundant Number Systems and a Variable Factor

In order to accelerate the CORDIC iterations, one can use redundant number systems, as we did for exponentials and logarithms in the previous chapter, but it is more difficult, because of the scale factor. With redundant representations, the main problem is the evaluation of d_n: the arithmetic operations themselves are quickly performed. Assume that we are in rotation mode. We want to evaluate d_n as quickly as possible. When performing the "nonredundant" iteration, d_n is equal to 1 if $z_n \geq 0$, and to -1 otherwise, where $z_{n+1} = z_n - d_n \arctan 2^{-n}$. Now we also allow the choice $d_n = 0$. The various functions $z_{n+1} = z_n - d_n \arctan 2^{-n}$ (for $d_n = -1, 0, 1$) are drawn on the Robertson diagram given in Figure 9.2.

The values r_n, A_n, and B_n in Figure 9.2 satisfy:

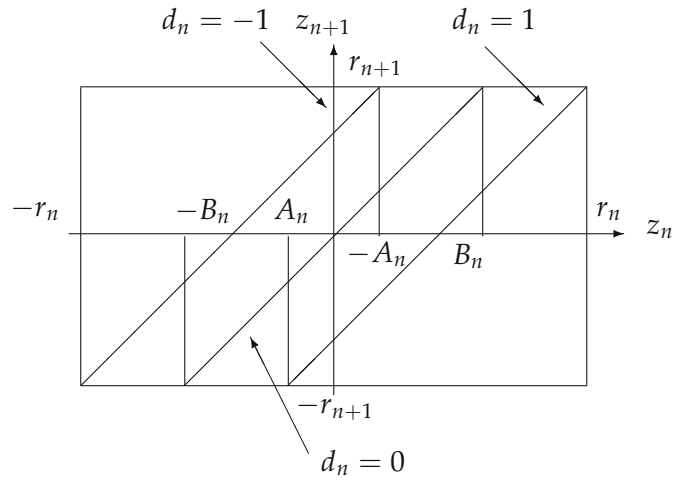


Figure 9.2 *Robertson diagram of CORDIC.*

Table 9.4 *First four values of* $2^n r_n$, $2^n A_n$ *and* $2^n B_n$.

n	$2^n r_n$	$2^n A_n$	$2^n B_n$
0	1.74328	–0.172490	0.957888
1	1.91577	–0.061186	0.988481
2	1.97696	–0.017134	0.997048
3	1.99409	–0.004417	0.999257
4	1.99851	–0.001113	0.999814
∞	2	0	1

$$\begin{cases} r_n &= \sum_{k=n}^{\infty} \arctan 2^{-k} \\ B_n &= r_{n+1} \\ A_n &= -r_{n+1} + \arctan 2^{-n}. \end{cases}$$

Table 9.4 gives the first four values of $2^n r_n$, $2^n A_n$, and $2^n B_n$. One can easily show that

$$\begin{aligned} A_n &\leq 0 \\ 2^n B_n &> 1/2. \end{aligned}$$

The Robertson diagram shows that

- if $z_n \leq -A_n$, then $d_n = -1$ is an allowable choice;
- if $-B_n \leq z_n \leq B_n$, then $d_n = 0$ is an allowable choice;
- if $z_n \geq A_n$, then $d_n = +1$ is an allowable choice.

Now let us see what would be obtained if we tried to implement this redundant CORDIC iteration in signed-digit or carry-save arithmetic.

9.5.1 Signed-Digit Implementation

Assume that we use the radix-2 signed-digit system to represent z_n. Assume that $-r_n \leq z_n \leq r_n$. Defining z_n^* as $2^n z_n$ truncated after its first fractional digit, we have

$$\left| z_n^* - 2^n z_n \right| \leq 1/2.$$

Therefore if we choose

$$d_n = \begin{cases} -1 \text{ if } z_n^* < 0 \\ 0 \text{ if } z_n^* = 0 \\ 1 \text{ if } z_n^* > 0, \end{cases} \tag{9.13}$$

then z_{n+1} is between $-r_{n+1}$ and r_{n+1}; this suffices for the algorithm to converge.

Proof

- If $z_n^* < 0$, then, since z_n^* is a 1-fractional digit number, $z_n^* \leq -1/2$; therefore $2^n z_n \leq 0$, and hence $z_n \leq -A_n$. Therefore (see Figure 9.2) choosing $d_n = -1$ will ensure $-r_{n+1} \leq z_{n+1} \leq r_{n+1}$.
- If $z_n^* = 0$, then $-1/2 \leq 2^n z_n \leq 1/2$; hence $-2^n B_n \leq 2^n z_n \leq 2^n B_n$; that is, $-B_n \leq z_n \leq B_n$. Therefore (see Figure 9.2) choosing $d_n = 0$ will ensure $-r_{n+1} \leq z_{n+1} \leq r_{n+1}$.
- If $z_n^* > 0$, with a similar deduction, one can show that choosing $d_n = +1$ will ensure $-r_{n+1} \leq z_{n+1} \leq r_{n+1}$.

9.5.2 Carry-Save Implementation

Assume now that we use the carry-save system for representing z_n. Define z_n^* as $2^n z_n$ truncated after its first fractional digit. We have

$$0 \leq 2^n z_n - z_n^* \leq 1;$$

therefore, if we choose

$$d_n = \begin{cases} -1 \text{ if } z_n^* < -1/2 \\ 0 \text{ if } z_n^* = -1/2 \\ 1 \text{ if } z_n^* > -1/2, \end{cases} \tag{9.14}$$

then z_n will be between $-r_{n+1}$ and r_{n+1}. The proof is similar to the proof for the signed-digit case.

9.5.3 The Variable Scale Factor Problem

Unfortunately, the "redundant methods" previously given cannot be easily used for the following reason. The scale factor K is equal to

$$\prod_{i=0}^{\infty} \sqrt{1 + d_k^2 2^{-2k}}.$$

If (as in the nonredundant CORDIC algorithm), $d_n = \pm 1$, then K is a constant. However, if d_n is allowed to be zero, then K is no longer a constant. Two classes of techniques have been proposed to overcome this drawback:

- one can compute the value of K (or, merely, $1/K$, so that the scale factor compensation is implemented by performing a multiplication instead of a division) *on the fly*, in parallel with the CORDIC iterations. This was suggested by Ercegovac and Lang [175, 178];
- one can modify the basic CORDIC iterations so that the scaling factor becomes a constant again.

This last solution is examined in the next sections.

9.6 The Double Rotation Method

This method was suggested independently by Takagi et al. [443, 444, 445] and by Delosme [149, 238], with different purposes. Takagi wanted to get a constant scaling factor when performing the iterations in a redundant number system, and Delosme wanted to perform simultaneously, in a conventional

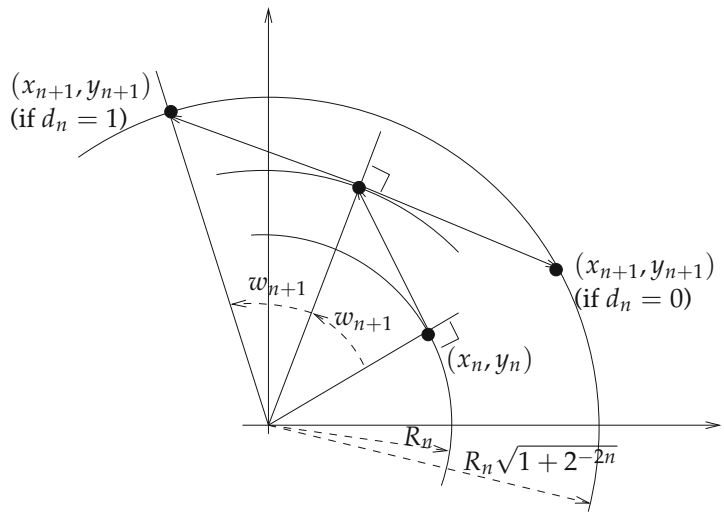


Figure 9.3 *One iteration of the double rotation method.*

number system, the vectoring operation and the rotation by half the resulting angle. The basic idea behind the double rotation method, illustrated in Figure 9.3 is to perform the similarities of angle $\pm \arctan 2^{-i}$ *twice*. Assume that we are in the circular ($m = 1$) type of iterations, in rotation mode. Instead of using the discrete base (see Chapter 8) $w_n = \arctan 2^{-n}$, we use the base

$$w'_n = 2 \arctan 2^{-n-1}.$$

Once d_n is found, the elementary similarity of angle $2d_n \arctan 2^{-n-1}$ is performed as follows:

- If $d_n = 1$, then we perform two similarities of angle $+\arctan 2^{-n-1}$;
- if $d_n = -1$, then we perform two similarities of angle $-\arctan 2^{-n-1}$;
- if $d_n = 0$, then we perform a similarity of angle $+\arctan 2^{-n-1}$, followed by a similarity of angle $-\arctan 2^{-n-1}$.

The basic iteration of the double rotation method becomes

$$\begin{cases} x_{n+1} = x_n - d_n 2^{-n} y_n + (1 - 2d_n^2) 2^{-2n-2} x_n \\ y_{n+1} = y_n + d_n 2^{-n} x_n + (1 - 2d_n^2) 2^{-2n-2} y_n \\ z_{n+1} = z_n - d_n w'_n = z_n - 2d_n \arctan 2^{-n-1}. \end{cases} \tag{9.15}$$

The new scaling factor is

$$K_{double} = \prod_{i=1}^{\infty} \left(1 + 2^{-2i}\right) = 1.3559096738634793803 \cdots .$$

The convergence domain is $I = [-A, +A]$, where

$$A = 2 \sum_{i=1}^{\infty} \arctan 2^{-i} = 1.91577691 \cdots .$$

The constant scale factor redundant CORDIC algorithm that uses the double rotation method was first suggested by N. Takagi in his Ph.D. dissertation [443]. It consists of performing (9.15) with the following choice of d_n.

$$d_n = \begin{cases} -1 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] < 0 \\ 0 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] = 0 \\ +1 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] > 0, \end{cases} \tag{9.16}$$

where $z_n = z_n^0.z_n^1 z_n^2 z_n^3 z_n^4 \cdots$. This algorithm works if we make sure that $z_n^{(n-1)}$ is the most significant digit of z_n. This is done by first noticing that

$$|z_n| \leq \sum_{k=n}^{\infty} \left(2 \arctan 2^{-k-1}\right) < 2^{-n+1}.$$

Define $\hat{z_n}$ as the number $0.0000 \cdots 0 z_n^{(n-2)} z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)} \cdots$ obtained by suppressing the possible digits of z_n of weight greater than 2^{-n+2}. The number $z_n - \hat{z}_n$ is a multiple of 2^{-n+3}, therefore

1. if $z_n^{(n-2)} = -1$, then
   - if $z_n^{(n-1)} = -1$, then $\hat{z}_n$ is between -2^{-n+3} and -2^{-n+2}; therefore the only possibility compatible with $|z_n| < 2^{-n+1}$ is $z_n = \hat{z}_n + 2^{-n+3}$. Thus we can rewrite z_n with the most significant digit at position $n-1$ by replacing $z_n^{(n-1)}$ by 1 and zeroing all the digits of z_n at the left of $z_n^{(n-1)}$ (of course, in practice, these digits are not "physically" zeroed: we just no longer consider them);
   - if $z_n^{(n-1)} = 0$, then $\hat{z}_n$ is between $-3 \times 2^{-n+1}$ and -2^{-n+1}; therefore there is no possibility compatible with $|z_n| < 2^{-n+1}$: this is an impossible case;
   - if $z_n^{(n-1)} = 1$, then $\hat{z}_n$ is between -2^{-n+2} and 0; therefore the only possibility compatible with $|z_n| < 2^{-n+1}$ is $z_n = \hat{z}_n$. Thus we can rewrite z_n by replacing $z_n^{(n-1)}$ by -1 and zeroing all the digits of z_n at the left of $z_n^{(n-1)}$.
2. if $z_n^{n-2} = 0$, then
   - if $z_n^{(n-1)} = -1$, then $\hat{z}_n$ is between -2^{-n+2} and 0; therefore $z_n = \hat{z}_n$. We do not need to modify $z_n^{(n-1)}$;
   - similarly, if $z_n^{(n-1)}$ is equal to 0 or 1, there is no need to modify it;
3. if $z_n^{(n-2)} = +1$, we are in a case that is similar (symmetrical, in fact) to the case $z_n^{(n-2)} = -1$.

Therefore it suffices to examine $z_n^{(n-2)}$: if it is nonzero, we negate $z_n^{(n-1)}$. After this, we can use (9.16) to find d_n using a small table. Of course, another solution is to directly use a larger table that gives d_n as a function of $[z_n^{(n-2)} z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}]$ as we did, for instance, in Section 8.3.1.

Another redundant CORDIC method with constant scale factor, suggested by Takagi et al. [445], is the *correcting rotation method*. With that method, at each step, d_n is chosen by examining a "window" of digits of z_n. If the window suffices to make sure that $z_n \geq 0$, then we choose $d_n = +1$, and if the window suffices to make sure that $z_n \leq 0$, then we choose $d_n = -1$. Otherwise, we choose d_n equal to $+1$ (a similar algorithm is obtained by choosing -1). By doing this, an error may be made, but that

error is small (because the fact that we are not able to find the sign of z_n implies that z_n is very small). One can show that this error can be corrected by repeating an extra iteration every m steps, where m can be an arbitrary integer (of course, the size of the "window" of digits depends on m).

9.7 The Branching CORDIC Algorithm

This algorithm was introduced by Duprat and Muller [168]. Corrections and improvements (a double-step branching CORDIC) have been suggested by Phatak [383, 384]. To implement the algorithm, we must assume that we have two "CORDIC modules," that is, that we are able to perform two conventional CORDIC iterations in parallel. The modules are named "module +" and "module −." The basic idea is the following: first we perform the conventional CORDIC iterations on one module; the only values of d_n that are allowed are -1 and $+1$. At step n, we try to estimate d_n by examining a "window" of digits of z_n. If this examination suffices to know the sign of z_n, then we choose $d_n = sign(z_n)$, as usual. Otherwise, we split the computations (this is what we call a "branching"): module "+" tries $d_n = +1$ and then continues to perform the conventional CORDIC iterations, and module "−" tries $d_n = -1$. If no module creates a new branching (i.e., if in both modules the "windows" of digits always suffice to estimate the sign of z_k), then both modules give a correct result. If a module creates a branching, say, at step m, this means that its value of z_m is very small, hence the choice of d_n tried by this module at the previous branching was a correct one. Therefore we can stop the computation that was performed by the other module, both modules are ready to carry on the computations needed by the new branching. This shows that even if many branchings are created, there is never any need for more than two modules.

A comparison of some variants of the Branching CORDIC algorithm is provided by Phatak in [462].

9.8 The Differential CORDIC Algorithm

This algorithm was introduced by Dawid and Meyr [130]. It allows a constant scale factor redundant implementation without additional operations.

First let us focus on the rotation mode. We start from an initial value z_0, $-\sum_{n=0}^{\infty} \arctan 2^{-n} \leq z_0 \leq +\sum_{n=0}^{\infty} \arctan 2^{-n}$, and the problem is to find, as fast as possible, values d_n, $d_n = \pm 1$, such that $z_0 = \sum_{n=0}^{\infty} d_n \arctan 2^{-n}$. We actually compute the values d_n that would have been given by the conventional algorithm (see Section 9.2), in a faster way. Since $d_n = \pm 1$, the scaling factor $K = \prod_{n=0}^{\infty} \sqrt{1 + d_n^2 2^{-2n}}$ remains constant.

Instead of the sequence z_n, we manipulate a new sequence $(\hat{z}_n)$, defined as $\hat{z}_{n+1} = \text{sign}(z_n) \times z_{n+1}$. From $d_n = \text{sign}(z_n)$ and $z_{n+1} = z_n - d_n \arctan 2^{-n}$, we find

$$z_{n+1} \times \text{sign}(z_n) = |z_n| - \arctan 2^{-n}.$$

Therefore

$$\begin{cases} |\hat{z}_{n+1}| = \left| |\hat{z}_n| - \arctan 2^{-n} \right| \\ d_{n+1} \quad = d_n \times \text{sign}\left(\hat{z}_{n+1}\right). \end{cases} \tag{9.17}$$

Assume that $\hat{z}_n$ is represented in the binary signed-digit number system (radix 2, digits -1, 0, and 1). Relations (9.17) allow us to partially separate the computation of the absolute value of $\hat{z}_{n+1}$ and the computation of its sign. To compute the absolute value of a binary signed-digit number $x = x_0.x_1x_2x_3x_4\cdots$ as well as its sign, we proceed *on-line* (i.e., in a digit-serial fashion, most significant

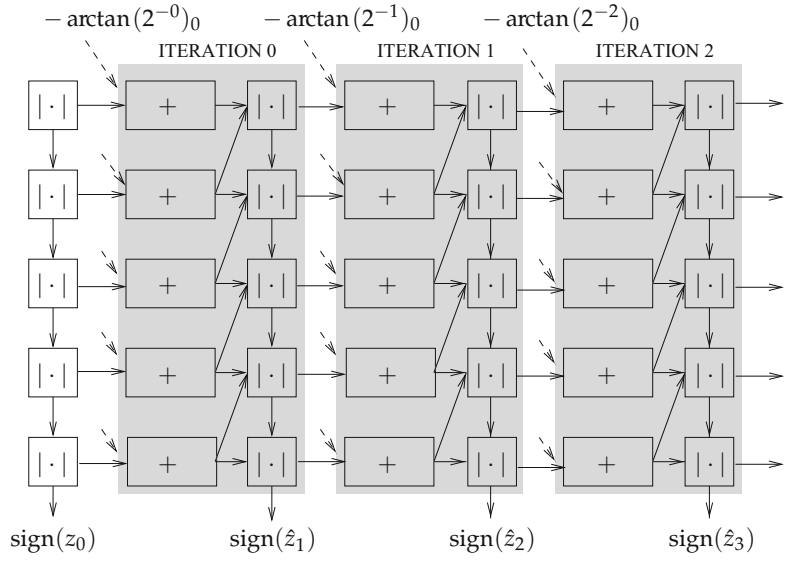


Figure 9.4 *Computation of the values* $\text{sign}(\hat{z}_i)$ *in the differential CORDIC algorithm (rotation mode) [130].*

digit first[5]). We examine its digits from left to right. If the first digits are equal to zero, we do not immediately know the sign of x, but we can output digits of $|x|$ anyway, it suffices to output zeroes. As soon as we see a nonzero digit x_i, we know the sign of x (it is the sign of that digit), and we can continue to output the digits of $|x|$. They are the digits of x if x is positive; otherwise, they are their negation.

Therefore we can implement iteration (9.17) in a digit-pipelined fashion: as soon as we get digits of $\left|\hat{z}_n\right|$, we can subtract (without carry propagation) $\arctan 2^{-n}$ from z_n and then generate the absolute value of the result to get digits of $\left|\hat{z}_{n+1}\right|$. $\left|\hat{z}_{n+1}\right|$ is generated from $\left|\hat{z}_n\right|$ with *on-line delay* 1: as soon as we get the ith digit of $\left|\hat{z}_n\right|$, we can compute the $i-1$st digit of $\left|\hat{z}_{n+1}\right|$. This is because adding a signed-digit binary number ($\left|\hat{z}_n\right|$) and a number represented in the conventional nonredundant number system ($-\arctan 2^{-n}$) can be done with on-line delay 1: the ith digit of the sum only depends on the ith and $i+1$st digits of the input operands (more details can be found, for instance, in [24]). This can be viewed in Figure 2.8.

The on-line delay required to get the sign — that is, to get a new value d_i using (9.17) — may be as large as the word length (if the only nonzero digit of a number is the least significant one, its sign is unknown until all the digits have been scanned), but it is clear from Figure 9.4 that this appears only once in the beginning. Using the operators depicted in that figure, the following algorithm makes it possible to perform the CORDIC rotations quickly, with constant scaling factor (the same factor as that of the conventional iteration).

[5]On-line arithmetic was introduced in 1977 by Ercegovac and Trivedi [182]. It requires the use of a redundant number system and introduces parallelism between sequential operations by overlapping them in a digit-serial fashion. See [174] for a survey.

Algorithm 18 Differential CORDIC, rotation mode

- input values: x_0, y_0 (input vector), z_0 (rotation angle),
- output values: x_{n+1}, y_{n+1} (scaled rotated vector).

$\hat{z}_0 = |z_0|$, $d_0 = sign\ (z_0)$
for $i = 0$ to n do

$$\begin{cases} \left|\hat{z}_{i+1}\right| &= \left|\left|\hat{z}_i\right| - \arctan 2^{-i}\right| \\ d_{i+1} &= d_i \times \text{sign}(\hat{z}_{i+1}) \\ x_{i+1} &= x_i - d_i y_i 2^{-i} \\ y_{i+1} &= y_i + d_i x_i 2^{-i}. \end{cases}$$

Dawid and Meyr also suggested a slightly more complex algorithm for the vectoring mode (i.e., for computing arctangents). As in the conventional vectoring mode, the iterations are driven by the sign of y_n, and that sign is computed using variables $\hat{x}_n$ and $\hat{y}_n$ defined by $\hat{x}_{n+1} = \text{sign}(x_n) \times x_{n+1}$ and $\hat{y}_{n+1} = \text{sign}(y_n) \times y_{n+1}$.

The relations allowing us to find the sign of y_n are

$$\begin{cases} \left|\hat{y}_{n+1}\right| &= \left|\left|\hat{y}_n\right| - \hat{x}_n 2^{-n}\right| \\ \text{sign}(y_{n+1}) &= \text{sign}(\hat{y}_{n+1}) \times \text{sign}(y_n). \end{cases}$$

They can be implemented using an architecture very similar to that of Figure 9.4. Using that architecture, the following algorithm performs the CORDIC iterations in vectoring mode with a constant scale factor:

Algorithm 19 Differential CORDIC, vectoring mode

- input values: x_0, y_0 (input vector), $z_0 = 0$,
- output values: x_{n+1} (scaled magnitude of the input vector), z_{n+1} (arctan y_0/x_0).

$\hat{x}_0 = x_0$, $\hat{y}_0 = |y_0|$
for $i = 0$ to n do

$$\begin{cases} \left|\hat{y}_{i+1}\right| &= \left|\left|\hat{y}_i\right| - \hat{x}_i 2^{-i}\right| \\ d_{i+1} &= d_i \times \text{sign}(\hat{y}_{i+1}) \\ \hat{x}_{i+1} &= \hat{x}_i + \left|\hat{y}_i 2^{-i}\right| \\ z_{i+1} &= z_i + d_i \arctan 2^{-i}. \end{cases}$$

The differential CORDIC algorithm can be extended to the hyperbolic mode in a straightforward manner. At first glance, it seems that Dawid and Meyrs' technique gives *nonrestoring* decompositions only; that is, it can be used to find decompositions $z_0 = \sum_{n=0}^{\infty} d_n w_n$ with $d_n = \pm 1$ only.

Yet it would be useful to generalize that technique to get decompositions with $d_n = 0, 1$ (i.e., "restoring decompositions"), since they would allow us to get an efficient algorithm for computing the exponential function[6] using $w_n = \ln\left(1 + 2^{-n}\right)$ (see Chapter 8). Such a generalization is simple. Assume that we want to get a restoring decomposition of a number x, $0 \leq x \leq \sum_{n=0}^{\infty} w_n$, that is, to get

$$x = \sum_{n=0}^{\infty} d_n w_n, \quad d_n = 0, 1. \tag{9.18}$$

Defining $S = \sum_{n=0}^{\infty} w_n$, this gives $2x - S = \sum_{n=0}^{\infty}(2d_n - 1)w_n$, that is,

$$2x - S = \sum_{n=0}^{\infty} \delta_n w_n, \quad \delta_n = \pm 1,$$

with $\delta_n = 2d_n - 1$.

Therefore, to get decomposition (9.18), it suffices to use the architecture described in Figure 9.4 with $2x - S$ as input. This gives values $\delta_n = \pm 1$, and each time we get a new δ_n, we deduce the corresponding term $d_n = 0, 1$ as $d_n = (\delta_n + 1)/2$.

9.9 The "CORDIC II" Approach

Garrido, Källström, Kumm, and Gustafsson [201] introduced another redundant CORDIC method with constant scale factor. The main idea is to consider iterations of the form

$$\begin{cases} \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} C_n & -d_n S_n \\ d_n S_n & C_n \end{pmatrix} \cdot \begin{pmatrix} x_n \\ y_n \end{pmatrix} \quad \text{with} \quad d_n = \pm 1, \\ \\ z_{n+1} = z_n - d_n \arctan\left(\dfrac{S_n}{C_n}\right), \end{cases} \tag{9.19}$$

where, at step n, C_n and S_n are taken from a "rotation kernel" $\mathcal{R}_n$ of possible pairs such that

- all terms C_n and S_n fit into a small number of bits (Garrido et al. present these terms as small integers),
- for all pairs $(C_n, S_n) \in \mathcal{R}_n$, $C_n^2 + S_n^2$ has the same value.

At step n, we perform a combination of a rotation of angle

$$\alpha_n = \pm \arctan(S_n / C_n),$$

[6] If we only wanted to compute exponentials, it would be simpler to implement than the hyperbolic mode of CORDIC. Of course, if we wished to design an architecture able to compute more functions, CORDIC might be preferred.

where $(S_n, C_n) \in \mathcal{R}_n$ and a multiplication by a factor

$$\sqrt{C_n^2 + S_n^2}$$

that does not depend on the choice of α_n.

Garrido et al. suggest to perform several "rotation stages":

- first, trivial rotations (that are true rotations, there is no scale factor) by an angle $\pm pi$ or $\pm\pi/2$, so that the remaining angle belongs to $[-\pi/4, +\pi/4]$;
- then, iterations of the form (9.19), with (C_n, S_n) taken in the rotation kernel

$$\{(25, 0), (24, 7), (20, 15)\}.$$

  One easily sees that these "rotations" (in fact, similarities) all lead to the same scale factor, since $25^2 + 0^2 = 24^2 + 7^2 = 20^2 + 15^2 = 625$;
- then, iterations of the form (9.19), with the rotation kernel

$$\{(129, 0), (128, 16)\};$$

- this can be continued with similar iterations. Garrido et al. cleverly notice that at some point we can use classic CORDIC iterations (or very simple iterations called by them "nano-rotations") since for small angles the variation of the scaling factor of the classic redundant CORDIC iteration becomes negligible (we can even perform scaling-free rotations, as in Section 9.3). See [201] for more details.

9.10 Computation of $\cos^{-1}$ and $\sin^{-1}$ Using CORDIC

Now we present another application [336] of the double rotation method introduced in Section 9.6. Assume that we want to compute $\theta = \cos^{-1} t$. When we perform a rotation of angle θ of the point $(1, 0)^t$ using CORDIC, we perform

$$\begin{cases} \theta_0 & = 0 \\ x_0 & = 1 \\ y_0 & = 0 \\ d_n & = \begin{cases} 1 \text{ if } \theta_n \leq \theta \\ -1 \text{ otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} & = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} & = \theta_n + d_n \arctan 2^{-n}, \end{cases} \tag{9.20}$$

and the sequence θ_n goes to θ as n goes to $+\infty$. Since the value of θ is not known (it is the value that we want to compute), we cannot perform the test

$$d_n = \begin{cases} 1 \text{ if } \theta_n \leq \theta \\ -1 \text{ otherwise} \end{cases} \tag{9.21}$$

that appears in Eq. (9.20). However, (9.21) is equivalent to:

$$d_n = \begin{cases} \text{sign}(y_n) \text{ if } \cos(\theta_n) \geq \cos(\theta) \\ -\text{sign}(y_n) \text{ otherwise,} \end{cases} \tag{9.22}$$

where $\text{sign}(y_n) = 1$ if $y_n \geq 0$, else -1. Since the variables x_n and y_n obtained at step n satisfy $x_n = K_n \cos\theta_n$ and $y_n = K_n \sin\theta_n$, where $K_n = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}}$, (9.22) is equivalent to

$$d_n = \begin{cases} \text{sign}(y_n) \text{ if } x_n \geq K_n t \\ -\text{sign}(y_n) \text{ otherwise.} \end{cases} \tag{9.23}$$

If we assume that the values $t_n = K_n t$ are known, Algorithm 20 below gives $\theta_n \to_{n\to\infty} \cos^{-1} t$.

Algorithm 20 $\cos^{-1}$: first attempt

$$\begin{cases} \theta_0 &= 0 \\ x_0 &= 1 \\ y_0 &= 0 \\ d_n &= \begin{cases} sign(y_n) \text{ if } x_n \geq t_n \\ -sign(y_n) \text{ otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} &= \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} &= \theta_n + d_n \arctan 2^{-n} \end{cases} \tag{9.24}$$

The major drawback of this algorithm is the need to know t_n. To compute t_n, the relation $t_{n+1} = t_n\sqrt{1+2^{-2n}}$ cannot be realistically used since it would imply a multiplication by $\sqrt{1+2^{-2n}}$ at each step of the algorithm. We overcome this drawback by performing double rotations: at each step of the algorithm, we perform *two* rotations of angle $d_n \arctan 2^{-n}$. In step n, the factor of the similarity becomes $1+2^{-2n}$; now a multiplication by this factor reduces to an addition and a shift. We obtain the following algorithm:

Algorithm 21 $\cos^{-1}$ with double-CORDIC iterations

$$\begin{cases} \theta_0 &= 0 \\ x_0 &= 1 \\ y_0 &= 0 \\ t_0 &= t \\ d_n &= \begin{cases} sign(y_n) \text{ if } x_n \geq t_n \\ -sign(y_n) \text{ otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} &= \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} &= \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} &= t_n + t_n 2^{-2n}. \end{cases} \tag{9.25}$$

The final value of θ_n is $\cos^{-1} t$ with an error close to 2^{-n}, for any $t \in [-1, 1]$. The next algorithm is similar.

Algorithm 22 $\sin^{-1}$ with double-CORDIC iterations

$$\begin{cases} \theta_0 & = 0 \\ x_0 & = 1 \\ y_0 & = 0 \\ t_0 & = t \\ d_n & = \begin{cases} sign(x_n) \text{ if } y_n \le t_n \\ -sign(x_n) \text{ otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} & = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} & = \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} & = t_n + t_n 2^{-2n}. \end{cases} \tag{9.26}$$

The final value of θ_n is $\sin^{-1} t$ with an error close to 2^{-n}.

Another method has been suggested by Lang and Antelo [291, 292] to compute $\sin^{-1}$ and $\cos^{-1}$ without the necessity of performing double rotations. This is done by building an approximation $A_n t$ to $K_n t$ that is good enough to ensure convergence if used in Equation (9.23) instead of $K_n t$ and easy to calculate.

9.11 Variations on CORDIC

Daggett [122] suggests the use of CORDIC for performing decimal-binary conversion. Schmid and Bogacki [412] suggest an adaptation of CORDIC to radix-10 arithmetic. Decimal CORDIC can be used in pocket calculators.[7] Decimal algorithms are briefly presented by Kropa [285]. Radix-4 CORDIC algorithms are suggested by Antelo et al. [10, 371]. An algorithm for very high radix CORDIC rotation was suggested later by Antelo, Lang, and Bruguera [11]. Shukla and Ray [424] obtain good performance by combining radix-4 iterations and the double-step branching CORDIC of Phatak [384].

A software implementation of CORDIC is reported by Andrews and Mraz [9]. CORDIC has been used for implementing the elementary functions in various coprocessors (such as the Intel 8087 and its successors until the 486, and the Motorola 68881). CORDIC was chosen for the 8087 because of its simplicity: the microcode size was limited to about 500 lines for the whole transcendental set [359]. Many other CORDIC chips are reported [106, 116, 153, 228, 438, 456, 478]. Some pipelined CORDIC architectures are suggested in [152, 276]. Adaptations of CORDIC to perform rotations in spaces of dimension higher than 2 are suggested by Delosme [150], Hsiao and Delosme [238], and Hsiao, Lau, and Delosme [237]. An angle recoding method that allows the reduction of the number of iterations when the angle of rotation is known in advance is suggested by Hu and Naganathan [241], and an improvement is suggested by Wu, Wu, and Lin [476]. Wang and Swartzlander [474] suggest to pair-off some iterations to lower the hardware complexity of a CORDIC processor. Timmermann et al. propose an algorithm [454] that is based on Baker's prediction method (see Section 8.4). Adaptations of CORDIC to on-line arithmetic were suggested by Ercegovac and Lang [176, 177, 178], Lin and Sips [319, 320], Duprat and Muller [168], and Hemkumar and Cavallaro [230]. Kota and Cavallaro [283] notice that in the vectoring mode of CORDIC small input values can result in large numerical errors, and they give methods to tackle this problem. Floating-point CORDIC algorithms have been suggested by Cavallaro and Luk [76], and by Hekstra and Deprettere [229]. An error analysis of CORDIC is

[7] Pocket calculators frequently use radix 10 for internal calculations and storage to avoid the radix conversion that would be required during inputs and outputs if they used radix 2.

given by Hu [240]. The survey by Meher et al. [338] should be read by anybody who wants to work on CORDIC or to implement it.

Some Other Shift-and-Add Algorithms 10

10.1 High-Radix Algorithms

The shift-and-add algorithms presented in Chapters 8 and 9 allow us to obtain an n-bit approximation of the function being computed after about n iterations. This property makes most of these algorithms rather slow; their major interest lies in the simplicity of implementation, and in the small silicon area of designs.

One can try to make these algorithms faster by implementing them in radices higher than 2: roughly speaking, a radix-2^k algorithm will only require n/k iterations to give an n-bit approximation of the function being computed. As for division and square root algorithms [179], the price to pay is more complicated elementary iterations.

10.1.1 Ercegovac's Radix-16 Algorithms

The methods presented here are similar to methods suggested by Ercegovac [171]. They are generalizable to higher radices, different from 16. Some variants have been proposed by Xavier Merrheim [342]. Assume that we want to compute the exponential of t. To do this, we use a basic iteration very close to (8.15); that is

$$\begin{aligned} L_{n+1} &= L_n - \ln\left(1 + d_n 16^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 16^{-n}\right), \end{aligned} \tag{10.1}$$

where the d_is belong to a "digit set" $\{-a, -a+1, \ldots, 0, 1, \ldots, a\}$, with $8 \leq a \leq 15$. Let us focus on the computation of the exponential function. One can easily see that if the d_is are selected such that L_n converge to 0, then E_n will converge to $E_0 \exp(L_0)$. As in Section 8.3.1, this is done by arranging that $L_n \in [s_n, r_n]$, where $s_n \to 0$ and $r_n \to 0$. Following Ercegovac [171], we choose $a = 10$. The Robertson diagram given in Figure 10.1 gives the various possible values of L_{n+1} versus L_n, depending on the choice of d_n.

One can easily show that the bounds s_n, r_n, s_{n+1}, and r_{n+1} that appear in the diagram of Figure 10.1 satisfy $r_{n+1} = r_n - \ln\left(1 + 10 \times 16^{-n}\right)$ and $s_{n+1} = s_n - \ln\left(1 - 10 \times 16^{-n}\right)$, which gives:

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_10

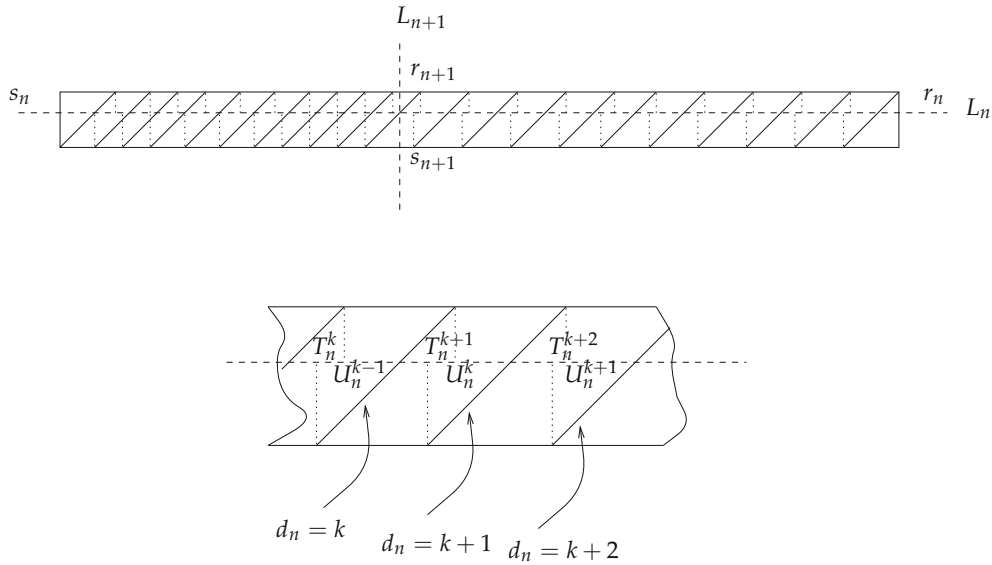


Figure 10.1 *Robertson diagram of the radix-16 algorithm for computing exponentials. T_k is the smallest value of L_n for which the value $d_n = k$ is allowable. U_k is the largest one.*

$$\begin{aligned} r_n &= \sum_{k=n}^{\infty} \ln\left(1 + 10 \times 16^{-k}\right) \\ s_n &= \sum_{k=n}^{\infty} \ln\left(1 - 10 \times 16^{-k}\right). \end{aligned} \tag{10.2}$$

For instance:

$$\begin{aligned} r_1 &= \sum_{n=1}^{\infty} \ln\left(1 + 10 \times 16^{-n}\right) \approx \quad 0.526427859674 \\ s_1 &= \sum_{n=1}^{\infty} \ln\left(1 - 10 \times 16^{-n}\right) \approx -1.023282325006. \end{aligned} \tag{10.3}$$

Define T_n^k as the smallest value of L_n such that the choice $d_n = k$ is allowable (i.e., such that L_{n+1} belong to $[s_{n+1}, r_{n+1}]$), and U_n^k as the largest one (see Figure 10.1). We find

$$\begin{aligned} T_n^k &= s_{n+1} + \ln\left(1 + k \times 16^{-n}\right) \\ U_n^k &= r_{n+1} + \ln\left(1 + k \times 16^{-n}\right). \end{aligned}$$

One can easily deduce from the Robertson diagram that if $U_n^k \geq T_n^{k+1}$ for any $k \in \{-10, \ldots, +9\}$, then for any $L_n \in [s_n, r_n]$ it is possible to find a value of d_n that is allowable. Moreover, if the values $U_n^k - T_n^{k+1}$ are large enough, the choice of d_n will be possible by examining a small number of digits of L_n only. Table 10.1 gives $16^n \times \min_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$ and $16^n \times \max_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$ for the first values of n.

One can see from Table 10.1 that the condition "$U_n^k \geq T_n^{k+1}$" is not satisfied for all values of k if $n = 1$. This means that the first step of the algorithm will have to differ from the following ones.

Define L_n^* as $16^n L_n$. We get

Table 10.1 *First four values of* $16^n \times \min_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$ *and* $16^n \times \max_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$, *and limit values for* $n \to \infty$.

n	$16^n \times \min_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$	$16^n \times \max_{k=-10\ldots9}\left(U_n^k - T_n^{k+1}\right)$
1	−1.13244	0.70644
2	0.29479	0.36911
3	0.33101	0.33565
4	0.33319	0.33348
∞	1/3	1/3

Table 10.2 *The interval* $16^n \times \left[T_n^k, U_n^k\right]$, *represented for various values of n and k. The integer k always belongs to that interval.*

	$n = 2$	$n = 3$	$n = \infty$
$k = -10$	[−10.87, −9.53]	[−10.68, −9.35]	$[-10 - \frac{2}{3}, -10 + \frac{2}{3}]$
$k = -9$	[−9.83, −8.496]	[−9.68, −8.34]	$[-9 - \frac{2}{3}, -9 + \frac{2}{3}]$
$k = -8$	[−8.80, −7.4618]	[−8.67, −7.34]	$[-8 - \frac{2}{3}, -8 + \frac{2}{3}]$
$k = -7$	[−7.76, −6.43]	[−7.67, −6.34]	$[-7 - \frac{2}{3}, -7 + \frac{2}{3}]$
$k = -6$	[−6.74, −5.41]	[−6.67, −5.34]	$[-6 - \frac{2}{3}, -6 + \frac{2}{3}]$
$k = -5$	[−5.72, −4.38]	[−5.67, −4.34]	$[-5 - \frac{2}{3}, -5 + \frac{2}{3}]$
$k = -4$	[−4.70, −3.37]	[−4.67, −3.34]	$[-4 - \frac{2}{3}, -4 + \frac{2}{3}]$
$k = -3$	[−3.69, −2.35]	[−3.67, −2.33]	$[-3 - \frac{2}{3}, -3 + \frac{2}{3}]$
$k = -2$	[−2.68, −1.34]	[−2.67, −1.33]	$[-2 - \frac{2}{3}, -2 + \frac{2}{3}]$
$k = -1$	[−1.67, −0.36]	[−1.67, −0.33]	$[-1 - \frac{2}{3}, -1 + \frac{2}{3}]$
$k = 0$	[−0.67, 0.67]	[−0.67, 0.67]	$[-\frac{2}{3}, +\frac{2}{3}]$
$k = 1$	[0.33, 1.66]	[0.33, 1.67]	$[1 - \frac{2}{3}, 1 + \frac{2}{3}]$
$k = 2$	[1.32, 2.66]	[1.33, 2.67]	$[2 - \frac{2}{3}, 2 + \frac{2}{3}]$
$k = 3$	[2.32, 3.65]	[2.33, 3.67]	$[3 - \frac{2}{3}, 3 + \frac{2}{3}]$
$k = 4$	[3.30, 4.63]	[3.33, 4.66]	$[4 - \frac{2}{3}, 4 + \frac{2}{3}]$
$k = 5$	[4.28, 5.62]	[4.33, 5.66]	$[5 - \frac{2}{3}, 5 + \frac{2}{3}]$
$k = 6$	[5.26, 6.60]	[5.33, 6.66]	$[6 - \frac{2}{3}, 6 + \frac{2}{3}]$
$k = 7$	[6.24, 7.57]	[6.33, 7.66]	$[7 - \frac{2}{3}, 7 + \frac{2}{3}]$
$k = 8$	[7.21, 8.5434]	[7.33, 8.66]	$[8 - \frac{2}{3}, 8 + \frac{2}{3}]$
$k = 9$	[8.18, 9.5113]	[8.32, 9.66]	$[9 - \frac{2}{3}, 9 + \frac{2}{3}]$
$k = 10$	[9.14, 10.48]	[9.32, 10.65]	$[10 - \frac{2}{3}, 10 + \frac{2}{3}]$

$$L_{n+1}^* = 16L_n^* - 16^{n+1} \ln\left(1 + d_n 16^{-n}\right). \tag{10.4}$$

The choice $d_n = k$ is allowable if and only if $16^n T_n^k \le L_n^* \le 16^n U_n^k$. One can show that for any $k \ge 2$,

$$16^n T_n^k < k < 16^n U_n^k.$$

This is illustrated by Table 10.2.

It is worth noticing that for $n \ge 3$, or $n = 2$ and $-8 \le k \le 8$, the interval where $d_n = k$ is a convenient choice (i.e., $[T_n^k, U_n^k]$), is much larger than the interval of the numbers that round to k (i.e., $[k - 1/2, k + 1/2]$). If $n \ge 3$, or $n = 2$ and $-8 \le k \le 8$, then

$$k + 1/2 + 1/32 < U_n^k$$

and

$$k - 1/2 - 1/32 > T_n^k.$$

This means that if $n \geq 3$, or $n = 2$ and $T_2^{-8} \leq L_2 \leq U_2^8$, d_n can be obtained by rounding to the nearest integer[1] either the number obtained by truncating the (nonredundant) binary representation of L_n^* after its fifth fractional digit, or the number obtained by truncating the carry-save representation of L_n^* after its sixth fractional digit.

There are two possibilities: First, we can start the iterations with $n = 2$; the convergence domain becomes $[T_2^{-8}, U_2^8] \approx [-0.03435, 0.03337]$, which is rather small. Or, we can implement a special first step. To do this, several solutions are possible. Assume that the input value x belongs to $[0, \ln(2)]$ (range reduction to this domain is fairly easy using the relation $\exp(x + k\ln(2)) = \exp(x) \times 2^k$), and compute from x a new initial value x^* and a correction factor M defined as follows. If x is between $k/16$ and $(k+1)/16$, then $x^* = x - (2k+1)/32$ belongs to $[T_2^{-8}, U_2^8]$, and $\exp(x)$ is obviously equal to $M \times \exp(x^*)$, where

$$M = e^{(2k+1)/32}.$$

[1] More precisely, to the nearest integer in $\{-10, \ldots, 10\}$ if $n \geq 3$; in $\{-8, \ldots, 8\}$ if $n = 2$.

Table 10.3 *Convenient values of ℓ for $x \in [0, \ln(2)]$. They are chosen such that $x - \ell \in [T_2^{-8}, U_2^8]$ and a multiplication by $\exp(\ell)$ is easily performed.*

$x \in$	ℓ
[0, 1/32]	0
[1/32, 3/32]	$\ln(1 + 2/32)$
[3/32, 4/32]	$\ln(1 + 4/32)$
[4/32, 5/32]	$\ln(1 + 5/32)$
[5/32, 6/32]	$\ln(1 + 6/32)$
[6/32, 7/32]	$\ln(1 + 7/32)$
[7/32, 8/32]	$\ln(1 + 9/32)$
[8/32, 9/32]	$\ln(1 + 10/32)$
[9/32, 10/32]	$\ln(1 + 11/32)$
[10/32, 11/32]	$\ln(1 + 13/32)$
[11/32, 12/32]	$\ln(1 + 14/32)$
[12/32, 14/32]	$\ln(1 + 16/32)$
[14/32, 15/32]	$\ln(1 + 19/32)$
[15/32, 16/32]	$\ln(1 + 20/32)$
[16/32, 17/32]	$\ln(1 + 22/32)$
[17/32, 18/32]	$\ln(1 + 24/32)$
[18/32, 20/32]	$\ln(1 + 26/32)$
[20/32, 21/32]	$\ln(1 + 29/32)$
[21/32, 22/32]	$\ln(1 + 31/32)$
[22/32, 23/32]	$\ln(2)$

There is a probably better solution, one that avoids the multiplication by a complicated factor M. It consists of finding from x a value ℓ such that $x^* = x - \ell$ belongs to $[T_2^{-8}, U_2^8]$ and a multiplication by $\exp(\ell)$ is easily reduced to a multiplication by a very small integer (which is easily performed using a small dedicated hardware) and a shift. In practice, we choose values of the form

$$\ell = \ln\left(1 + \frac{k}{32}\right), k = 1 \ldots 32,$$

and we have

$$e^x = e^{x^*}\left(1 + \frac{k}{32}\right).$$

Convenient values of ℓ are given in Table 10.3.

10.2 The BKM Algorithm

As shown in Chapter 9, CORDIC allows the computation of many functions. For this reason, it has been implemented in many pocket calculators and floating-point coprocessors. Its major drawback arises when performing the iterations using a *redundant* (e.g., carry-save or signed-digit) number system: to make the evaluation of d_n easier (see Chapter 9), we must accept $d_n = 0$, and this implies a nonconstant scale factor K, unless we accept performing more iterations, or more complicated iterations, or unless

we use Dawid and Meyr's method. We now examine an algorithm due to Bajard et al. [25], that allows us to perform rotations and to compute complex functions without scaling factors.

10.2.1 The BKM Iteration

In the following, we assume a radix-2 conventional or signed-digit number system. Extension to binary carry-save representation is straightforward. Let us consider the basic step of CORDIC in trigonometric mode (i.e., iteration (9.1)). If we define a complex number E_n as $E_n = x_n + iy_n$, we obtain $E_{n+1} = E_n\left(1 + id_n 2^{-n}\right)$, which is close to the basic step of Algorithm 8.1. This remark brings us to a generalization of that algorithm: we could perform multiplications by terms of the form $\left(1 + d_n 2^{-n}\right)$, where the d_ns are *complex numbers*, chosen such that a multiplication by d_n can be reduced to a few additions. In the following, we study the iteration:

$$\begin{cases} E_{n+1} = E_n\left(1 + d_n 2^{-n}\right) \\ L_{n+1} = L_n - \ln\left(1 + d_n 2^{-n}\right) \end{cases} \tag{10.5}$$

with $d_n \in \{-1, 0, 1, -i, i, 1-i, 1+i, -1-i, -1+i\}$.

We define $\ln z$ as the number t such that $e^t = z$, and whose imaginary part is between $-\pi$ and π. Exactly as in Chapter 8

- If we are able to find a sequence d_n such that E_n goes to 1, then we will obtain $L_n \to L_1 + \ln(E_1)$. This iteration mode is the *L-mode* of the algorithm.
- If we are able to find a sequence d_n such that L_n goes to 0, then we will obtain $E_n \to E_1 e^{L_1}$. This is the *E-mode* of the algorithm.

10.2.2 Computation of the Exponential Function (E-mode)

As pointed out at the end of the previous section, to compute e^{L_1} using BKM, one must find a sequence $d_n, d_n = -1, 0, 1, -i, i, i-1, i+1, -i-1, -i+1$, such that $\lim_{n\to\infty} L_n = 0$. Defining d_n^x and d_n^y as the real and imaginary parts of d_n (they belong to $\{-1, 0, 1\}$) and L_n^x and L_n^y as the real and imaginary parts of L_n, we find

$$\begin{cases} L_{n+1}^x = L_n^x - \dfrac{1}{2}\ln\left[1 + d_n^x 2^{-n+1} + \left(d_n^{x\,2} + d_n^{y\,2}\right)2^{-2n}\right] \\ L_{n+1}^y = L_n^y - d_n^y \arctan\left(\dfrac{2^{-n}}{1 + d_n^x 2^{-n}}\right). \end{cases} \tag{10.6}$$

Now we give an algorithm that computes the sequence d_n for any L_1 belonging to a rectangular set $R_1 = \left[-s_1^x, r_1^x\right] + i\left[-r_1^y, r_1^y\right]$. The proof of the algorithm is based on the construction of a sequence $R_n = \left[-s_n^x, r_n^x\right] + i\left[-r_n^y, r_n^y\right]$ of rectangular sets that contain zero, whose length goes to zero as n goes to infinity, and such that for any $L_n \in R_n$, d_n ensures that $L_{n+1} \in R_{n+1}$. The "real part" d_n^x is chosen by examining a few digits of L_n^x and the "imaginary part" d_n^y is chosen by examining a few digits of L_n^y. These properties allow a simple and fast implementation of the choice of d_n.

10.2.2.1 Choice of d_n^x

The Robertson diagram presented in Figure 10.2 shows the different parameters involved in determining d_n^x. The diagram is constructed as follows.

- We assume that L_n^x belongs to the interval $\left[-s_n^x, r_n^x\right]$, which is the real part of R_n.
- L_{n+1}^x is equal to $L_n^x - (1/2)\ln\left[1 + d_n^x 2^{-n+1} + \left(d_n^{x\,2} + d_n^{y\,2}\right)2^{-2n}\right]$, so the value of L_{n+1}^x versus L_n^x is given by various straight lines parameterized by d_n^x and d_n^y.
- d_n^x must be such that *for any possible value of* d_n^y, $L_{n+1}^x \in \left[-s_{n+1}^x, r_{n+1}^x\right]$.

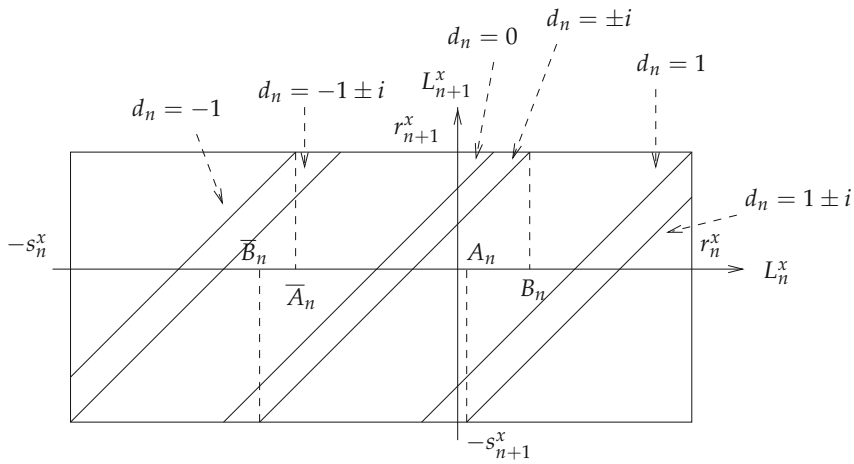


Figure 10.2 *The Robertson diagram for* L_n^x *[25].*

Parameter r_n^x is equal to

$$\sum_{k=n}^{\infty} \ln\left(1 + 2^{-k}\right),$$

and s_n^x is equal to

$$-\frac{1}{2}\sum_{k=n}^{\infty} \ln\left(1 - 2^{-k+1} + 2^{-2k+1}\right).$$

The terms $\bar{A}_n$, A_n, $\bar{B}_n$, and B_n appearing in the diagram shown in Figure 10.2 are

- $\bar{A}_n = r_{n+1}^x + \ln(1 - 2^{-n})$;
- $\bar{B}_n = -s_{n+1}^x + (1/2)\ln(1 + 2^{-2n})$;
- $A_n = -s_{n+1}^x + (1/2)\ln(1 + 2^{-n+1} + 2^{-2n+1})$;
- $B_n = r_{n+1}^x$.

One can prove that $\bar{B}_n < \bar{A}_n$, and that $A_n < B_n$. From this, for any $L_n^x \in \left[-s_n^x, r_n^x\right]$, these choices will give a value of L_{n+1}^x between $-s_{n+1}^x$ and r_{n+1}^x

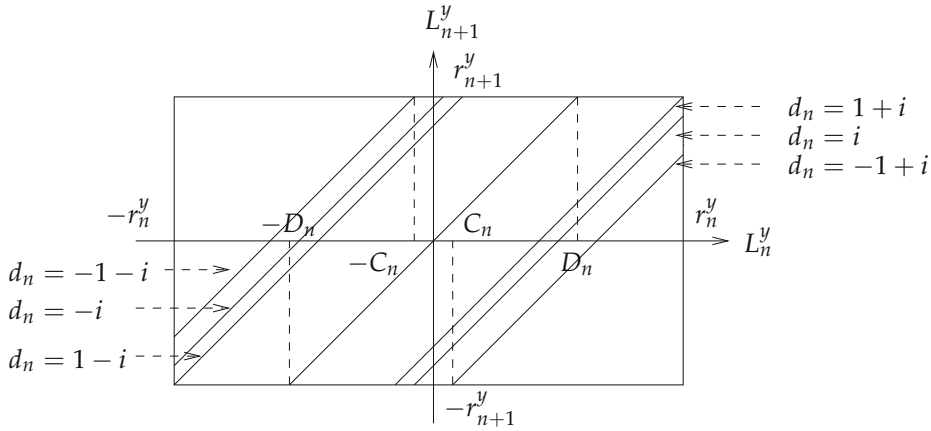


Figure 10.3 *The Robertson diagram for L^y_n [25].*

$$\begin{cases} \text{if} \quad L^x_n < -\bar{B}_n \quad \text{then} \ d^x_n = -1 \\ \text{if} \ -\bar{B}_n \le L^x_n < \bar{A}_n \ \text{then} \ d^x_n = -1 \text{ or } 0 \\ \text{if} \ \ \bar{A}_n \le L^x_n < A_n \ \text{then} \ d^x_n = 0 \\ \text{if} \ \ A_n \le L^x_n \le B_n \ \text{then} \ d^x_n = 0 \text{ or } 1 \\ \text{if} \quad B_n < L^x_n \quad \text{then} \ d^x_n = 1. \end{cases} \tag{10.7}$$

10.2.2.2 Choice of d^y_n

We use the relation

$$L^y_{n+1} = L^y_n - d^y_n \arctan\left(\frac{2^{-n}}{1 + d^x_n 2^{-n}}\right).$$

Figure 10.3 shows the Robertson diagram associated with the choice of d^y_n. We want our choice to be independent of the choice of d^x_n. From this, we deduce:

$$r^y_n = \sum_{k=n}^{\infty} \arctan\left(\frac{2^{-k}}{1 + 2^{-k}}\right).$$

The terms C_n and D_n appearing in the diagram are

$$C_n = -r^y_{n+1} + \arctan\left(\frac{2^{-n}}{1 - 2^{-n}}\right)$$

and

$$D_n = r^y_{n+1}.$$

One can prove that $C_n < D_n$. Thus, for any $L^y_n \in \left[-r^y_n, r^y_n\right]$, these choices will give a value of L^y_{n+1} between $-r^y_{n+1}$ and $+r^y_{n+1}$

$$\begin{cases} \text{if} & L_n^y < -D_n & \text{then } d_n^y = -1 \\ \text{if} & -D_n \le L_n^y < -C_n & \text{then } d_n^y = -1 \text{ or } 0 \\ \text{if} & -C_n \le L_n^y < C_n & \text{then } d_n^y = 0 \\ \text{if} & C_n \le L_n^y \le D_n & \text{then } d_n^y = 0 \text{ or } 1 \\ \text{if} & D_n < L_n^y & \text{then } d_n^y = 1. \end{cases} \tag{10.8}$$

The convergence domain R_1 of the algorithm is

$$\begin{aligned} -0.8298023738\cdots = -s_1^x \le L_1^x \le r_1^x = 0.8688766517\cdots \\ -0.749780302\cdots = -r_1^y \le L_1^y \le r_1^y = 0.749780302\cdots . \end{aligned}$$

10.2.2.3 The Algorithm

Relations (10.7) and (10.8) make it possible to find a sequence d_n such that, for $L_1 \in R_1$, $\lim_{n\to\infty} L_n = 0$. Now let us try to simplify the choice of d_n: (10.7) and (10.8) involve comparisons that may require the examination of all the digits of the variables; we want to replace these comparisons by the examination of a small number of digits. The parameters $\bar{A} = -1/2$, $A = 1/4$, $C = 3/4$, $p_1 = 3$, and $p_2 = 4$ satisfy, for every n:

$$\begin{cases} 2^n \bar{B}_n \le \bar{A} - 2^{-p_1} < \bar{A} \le 2^n \bar{A}_n \\ 2^n A_n \le A < A + 2^{-p_1} \le 2^n B_n \\ 2^n C_n \le C < C + 2^{-p_2} \le 2^n D_n. \end{cases} \tag{10.9}$$

Therefore if we call $\tilde{L}_n^x$ the number obtained by truncating $2^n L_n^x$ after its p_1th fractional digit, and $\tilde{L}_n^y$ the number obtained by truncating $2^n L_n^y$ after its p_2th fractional digit, we obtain, from (10.7) through (10.9)

- if $\tilde{L}_n^x \le \bar{A} - 2^{-p_1}$, then $L_n^x \le \bar{A}_n$; therefore $d_n^x = -1$ is a valid choice;
- if $\bar{A} \le \tilde{L}_n^x \le A$, then $\bar{B}_n \le L_n^x \le B_n$; therefore $d_n^x = 0$ is a valid choice[2];
- if $A + 2^{-p_1} \le \tilde{L}_n^x$, then $A_n \le L_n^x$; therefore $d_n^x = 1$ is a valid choice;
- if $\tilde{L}_n^y \le -C - 2^{-p_2}$, then $L_n^y \le -C_n$; therefore $d_n^y = -1$ is a valid choice;
- if $-C \le \tilde{L}_n^y \le C$, then $-D_n \le L_n^y \le D_n$; therefore $d_n^y = 0$ is a valid choice;
- if $C + 2^{-p_2} \le \tilde{L}_n^y$, then $C_n \le L_n^y$; therefore $d_n^y = 1$ is a valid choice.

10.2.2.4 Number of Iterations

After n iterations of the E-mode, we obtain a relative error approximately equal to 2^{-n}.

10.2.3 Computation of the Logarithm Function (L-mode)

Computing the logarithm of a complex number E_1 using the L-mode requires the calculation of a sequence d_n, $d_n = -1, 0, 1, -i, i, i-1, i+1, -i-1, -i+1$, such that

$$\lim_{n\to\infty} E_n = 1. \tag{10.10}$$

The following algorithm had been found through simulations before being proved. See [25] for more details.

[2]Since $\bar{A}$,A, and $\tilde{L}_n^x$ have at most p_1 fractional digits, if $\tilde{L}_n^x > \bar{A} - 2^{-p_1}$, then $\tilde{L}_n^x \ge \bar{A}$.

10.2.3.1 BKM Algorithm — L-mode

- Start with E_1 belonging to the trapezoid T delimited by the straight lines $x = 1/2$, $x = 1.3$, $y = x/2$, $y = -x/2$. T is the domain where the convergence is proven, but experimental tests show that the actual convergence domain of the algorithm is larger.
- Iterate $\begin{cases} E_{n+1} = E_n\left(1 + d_n 2^{-n}\right) \\ L_{n+1} = L_n - \ln\left(1 + d_n 2^{-n}\right) \end{cases}$
  with $d_n = d_n^x + i d_n^y$ chosen as follows:
  - define ϵ_n^x and ϵ_n^y as the real and imaginary parts of

$$\epsilon_n = 2^n(E_n - 1)$$

    and $\tilde{\epsilon}_n^x$ and $\tilde{\epsilon}_n^y$ as the values obtained by truncating these numbers after their fourth fractional digits;
  - at step 1

$$\begin{cases} \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } 6/16 \le \tilde{\epsilon}_1^y \text{ then } d_1 = 1 - i \\ \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } \tilde{\epsilon}_1^y \le -6/16 \text{ then } d_1 = 1 + i \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } 8/16 \le \tilde{\epsilon}_1^y \text{ then } d_1 = -i \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } \tilde{\epsilon}_1^y \le -9/16 \text{ then } d_1 = i \\ \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } \left|\tilde{\epsilon}_1^y\right| \le 5/16 \text{ then } d_1 = 1 \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } \left|\tilde{\epsilon}_1^y\right| \le 1/2 \text{ then } d_1 = 0; \end{cases}$$

  - at step n, $n \ge 2$

$$\begin{cases} \text{if } \tilde{\epsilon}_n^x \le -1/2 \text{ then } d_n^x = 1 \\ \text{if } -1/2 < \tilde{\epsilon}_n^x < 1/2 \text{ then } d_n^x = 0 \\ \text{if } 1/2 \le \tilde{\epsilon}_n^x \text{ then } d_n^x = -1 \end{cases}$$

$$\begin{cases} \text{if } \tilde{\epsilon}_n^y \le -1/2 \text{ then } d_n^y = 1 \\ \text{if } -1/2 < \tilde{\epsilon}_n^y < 1/2 \text{ then } d_n^y = 0 \\ \text{if } 1/2 \le \tilde{\epsilon}_n^y \text{ then } d_n^y = -1; \end{cases}$$

- result: $\lim_{n\to\infty} L_n = L_1 + \ln(E_1)$.

In a practical implementation, instead of computing E_n and examining the first digits of $\epsilon_n = 2^n(E_n - 1)$, one would directly compute the sequence ϵ_n. See [25] for a proof of the algorithm.

10.2.4 Application to the Computation of Elementary Functions

As shown in the previous sections, the algorithm makes it possible to compute the functions

- in E-mode, $E_1 e^{L_1}$, where L_1 is a complex number belonging to R_1,
- in L-mode, $L_1 + \ln(E_1)$, where E_1 belongs to the trapezoid T.

Therefore one can compute the following functions of real variables:

10.2.4.1 Functions Computable Using One Mode of BKM

- **Real sine and cosine functions**. In the E-mode, one can compute the exponential of $L_1 = 0 + i\theta$ and obtain
$$E_n = \cos\theta + i\sin\theta \pm 2^{-n}.$$
- **Real exponential function**. If L_1 is a real number belonging to $[-0.8298023738, +0.8688766517]$, the E-mode will give a value E_n equal to
$$E_1 e^{L_1} \pm 2^{-n}.$$
- **Real logarithm**. If E_1 is a real number belonging to $[0.5, 1.3]$, the L-mode will give a value L_n equal to
$$L_1 + \ln(E_1) \pm 2^{-n}.$$
- **2-D rotations**. The 2-D vector $(c\ d)^t$ obtained by rotating $(a\ b)^t$ by an angle θ satisfies: $c + id = (a + ib)e^{i\theta}$; therefore $(c\ d)^t$ is computed using the E-mode, with $E_1 = a + ib$ and $L_1 = i\theta$.
- **real** $arctan$ **function**. From the relation
$$\ln(x+iy) = \begin{cases} \dfrac{1}{2}\ln(x^2+y^2) + i\arctan\dfrac{y}{x} \bmod (2i\pi) & (x > 0) \\ \dfrac{1}{2}\ln(x^2+y^2) + i\left(\pi + \arctan\dfrac{y}{x}\right) \bmod (2i\pi) & (x < 0) \end{cases}$$
one can easily deduce that, if $x + iy$ belongs to the convergence domain of the L-mode, then $\arctan y/x$ is the limit value of the imaginary part of L_n, assuming that the L-mode is used with $L_1 = 0$ and $E_1 = x + iy$. The same operation also gives $\ln(x^2 + y^2)/2$.

10.2.4.2 Functions Computable Using Two Consecutive Modes of BKM

Using two BKM operations, one can compute many functions. Some of these functions are

- **Complex multiplication and division**. The product zt is evaluated as $z.e^{\ln t}$, and z/t is evaluated as $z.e^{-\ln t}$. One can compute $(ab)e^z$ or $(a/b)\,e^z$, where a, b, and z are complex numbers, using the same operator, by choosing L_1^x equal to the real part of z, and L_1^y equal to the imaginary part of z.
- **Computation of** $x\sqrt{a}$ **and** $y\sqrt{a}$ **in parallel** (x, y and a are *real* numbers): we use the relation $\sqrt{a} = e^{\ln(a)/2}$. One can also compute $x/\sqrt{a}$ and $y/\sqrt{a}$.
- **Computation of lengths and normalization of 2D-vectors**. As shown previously, the L-mode allows the computation of $F = \ln(a^2 + b^2)/2 = \ln\sqrt{a^2 + b^2}$, where a and b are real numbers. Using the E-mode, we can compute $x \cdot e^F$ or $x \cdot e^{-F}$.

A generalization of BKM to radix 10 arithmetic was suggested by Imbert, Muller, and Rico [246]. A High-Radix version of BKM was described by Didier and Rico [157].

Part III

Range Reduction, Final Rounding and Exceptions

Range Reduction

11

11.1 Introduction

The algorithms presented in the previous chapters for evaluating the elementary functions only give correct result if the argument is within a given bounded interval. To evaluate an elementary function $f(x)$ for any x, two cases may occur

- either there is some algebraic relation (such as $e^{a+b} = e^a \cdot e^b$, or the trigonometric formulas), that allows us to easily deduce the values of the function in the whole domain from its values in a small interval: in that case, we can reduce our initial problem to computations in that small interval;
- or there is not such a simple algebraic relation: in that case, we must split the input domain into several (possibly many!) subdomains, and use a different method (or a different polynomial or rational approximation) in each subinterval. If the input domain is very large, we will have to use asymptotic expansions—if there exist such usable expansions for the considered function—on both ends so that the remaining domain becomes of reasonable size. Once we have built approximations for each subinterval, the first part of the code that will implement the function will consist of branchings that determine in which domain the input value lies. Recently, Kupriianova and Lauter suggested an ingenious approach [290] for avoiding these branchings. The index of the interval where the input value lies is a function g of that input value. They try to approximate g itself by a polynomial. That method does not always work (and it can certainly be improved); but when it works, it considerably eases the design of vectorizable implementations.

Fortunately, with the classical elementary functions, we are almost always in the first case, and we will deal with that case in the following.

One must find some "transformation," deduced from some algebraic relation satisfied by the function f that makes it possible to deduce $f(x)$ from some value $g(x^*)$, where

- x^*, called the *reduced argument*, is deduced from x;
- x^* belongs to the convergence domain of the algorithm implemented for the evaluation of g.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_11

In practice, there are two kinds of reduction

- *Additive reduction.* x^* is equal to $x - kC$, where k is an integer and C is a constant (for instance, for the trigonometric functions, C is a multiple of $\pi/4$).
- *Multiplicative reduction.* x^* is equal to x/C^k, where k is an integer and C is a constant (for instance, for the logarithm function, a convenient choice for C is the radix of the number system).

Example 11 (cosine function) *We want to evaluate* $\cos(x)$, *and the convergence domain of the algorithm used to evaluate the sine and cosine of the reduced argument contains* $[-\pi/4, +\pi/4]$. *We choose* $C = \pi/2$ *and the computation of* $\cos(x)$ *is decomposed in three steps*

- *compute* x^* *and* k *such that* $x^* \in [-\pi/4, +\pi/4]$ *and* $x^* = x - k\pi/2$;
- *compute* $g(x^*, k) =$

$$\begin{cases} \cos(x^*) \text{ if } k \bmod 4 = 0 \\ -\sin(x^*) \text{ if } k \bmod 4 = 1 \\ -\cos(x^*) \text{ if } k \bmod 4 = 2 \\ \sin(x^*) \text{ if } k \bmod 4 = 3; \end{cases} \tag{11.1}$$

- *obtain* $\cos(x) = g(x^*, k)$.

The previous reduction mechanism is an *additive* range reduction. Let us examine another example of additive reduction.

Example 12 (exponential function) *We want to evaluate* e^x *in a radix-2 number system, and the convergence domain of the algorithm used to evaluate the exponential of the reduced argument contains* $[0, \ln(2)]$. *We can choose* $C = \ln(2)$, *and the computation of* e^x *is then decomposed in three steps*

- compute $x^* \in [0, \ln(2)]$ and k such that $x^* = x - k\ln(2)$;
- compute $g(x^*) = e^{x^*}$;
- compute $e^x = 2^k g(x^*)$.

The radix-2 number system makes the final multiplication by 2^k *straightforward.*

There are other ways of performing the range reduction for the exponential function. A solution (with an algorithm whose convergence domain is $[0, 1]$) is to choose $x^* = x - \lfloor x \rfloor, k = \lfloor x \rfloor$, and $g(x^*) = e^{x^*}$. Then $e^x = g(x^*) \times e^k$, and e^k can either be evaluated by performing a few multiplications—since k is an integer—or by table lookup. In one of his table-driven algorithms, Tang [450] uses

$$C = \frac{\ln(2)}{32}$$

so that the argument is reduced to a small interval

$$\left[-\frac{\ln(2)}{64}, +\frac{\ln(2)}{64}\right].$$

In any case, range reduction is more a problem for trigonometric functions than for exponentials, since, in practice, we never have to deal with exponentials of very large numbers: they are overflows! For instance, in IEEE-754 binary64/double-precision arithmetic and assuming round-to-nearest, when computing e^x

- if $x \leq \ln\left(2^{-1075}\right) \approx -745.13321910$ then 0 must be returned;
- if $x \geq \ln\left[\left(2 - 2^{-53}\right) \cdot 2^{1023}\right] \approx 709.78271289$ then $+\infty$ must be returned.

Example 13 (logarithm function) *We want to evaluate* $\ln(x)$, $x > 0$, *in a radix-2 number system, and the convergence domain of the algorithm used to compute the logarithm of the reduced argument contains* $[1/2, 1]$. *We can choose* $C = 2$, *and the computation of* $\ln(x)$ *is then decomposed in three steps*

- *compute* $x^* \in [1/2, 1]$ *and* k *such that* $x^* = x/2^k$ *(if* x *is a normalized radix-2 floating-point number,* x^* *is its significand, and* k *is its exponent);*
- *compute* $g(x^*, k) = \ln(x^*)$;
- *compute* $\ln(x) = g(x^*, k) + k \ln(2)$.

The previous mechanism is a multiplicative *reduction*.

In practice, multiplicative reduction is not a difficult problem: when computing the usual functions, it only occurs with logarithms and nth roots. With these functions, as in the preceding example, C can be chosen equal to a power of the radix of the number system. This makes the computation of x/C^k straightforward and errorless. Therefore, in the following, we concentrate on the problem of *additive* range reduction only.

It is worth noticing that, whereas the original argument x is a "machine number," the computed reduced argument x^* should, in general, *not* be a machine number: depending on the kind of implementation, if we aim at a very accurate result, it should be represented in a larger format, or by a double-word or triple-word number (see Section 2.2.2). This must be taken into account when designing an algorithm for evaluating the approximation.

A poor range reduction method may lead to catastrophic accuracy problems when the input arguments are large. Table 11.1 gives the computed value of $\sin(10^{22})$ on various computing systems (the figures in that table were picked up from [362, 363] and from various contributors belonging to the Computer Science Department of École Normale Supérieure de Lyon. They are rather old, and the behavior of modern systems is, in general, much better). Some results are very bad. Returning a NaN is probably better than returning a totally inaccurate result; however, this is not the right solution. Unless we design a special-purpose system, for which particular range and accuracy requirements may be known, we must always return results as close as possible to the exact values. On some machines, what was actually approximated was not the sine function, but the function

$$\mathrm{fsin}(x) = \sin\left(\frac{\pi x}{\hat{\pi}}\right)$$

where $\hat{\pi}$ is the floating-point number that is closest to π (for instance, in Table 11.1, the number $0.8740\cdots$ returned by the Silicon Graphics Indy computer was equal to $\mathrm{fsin}(10^{22})$). This of course made range reduction easier, but there were strange "side effects": when x is so small that no range reduction is necessary, returning the floating-point number that is closest to $\mathrm{fsin}(x)$ is not equivalent to returning the floating-point number that is closest to $\sin(x)$. For instance, in IEEE-754 binary64 arithmetic, the machine number that is closest to $\mathrm{fsin}(1/4)$ is

$$\frac{1114208378708655}{4503599627370496}$$

whereas the machine number that is closest to $\sin(1/4)$ is

Table 11.1 $\sin(x)$ *for* $x = 10^{22}$ *on various (in general, old) systems [362]. It is worth noticing that* x *is exactly representable in the IEEE-754 binary64 format (*10^{22} *is equal to* $6368371582031250 \times 2^{21}$*). With a system working in the IEEE-754 binary32 format, the correct answer would be the sine of the floating-point number that is closest to* 10^{22}*; that is,* $\sin(9999999778196308361216) \approx -0.73408$. *As pointed out by the authors of [362, 363], the values listed in this table were contributed by various Internet volunteers. Hence, they are not the official views of the listed computer system vendors, the author of [362, 363] or his employer, nor are they those of the author of this book.*

Computing System	$\sin x$
Exact result	$-0.8522008497671888017727\cdots$
Vax VMS (g or h format)	$-0.852200849\cdots$
HP 48 GX	-0.852200849762
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\cdots$
matlab V.4.2 c.1 for Macintosh	0.8740
matlab V.4.2 c.1 for SPARC	-0.8522
Silicon Graphics Indy	$0.87402806\cdots$
SPARC	-0.85220084976718879
IBM RS/6000 AIX 3005	$-0.852200849\cdots$
IBM 3090/600S-VF AIX 370	0.0
PC: Borland TurboC 2.0	$4.67734e-240$
Sharp EL5806	-0.090748172
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

$$\frac{4456833514834619}{18014398509481984}.$$

The difference between both values is equal to 1 ulp.

Even when x is not large, if it is close to a multiple of $\pi/4$, then a poor range reduction may lead to a very inaccurate evaluation of $\sin(x)$ or $\tan(x)$.

It is easy to understand why a bad range reduction algorithm gives inaccurate results. The naive method consists of performing the computations

$$\begin{aligned} k &= \left\lfloor \frac{x}{C} \right\rfloor \\ x^* &= \mathrm{RN}(x - \mathrm{RN}(kC)) \quad \text{or } \mathrm{RN}(x - kC) \text{ if an FMA is available} \end{aligned}$$

using the machine precision, where RN is the round-to-nearest rounding function. When kC is close to x, almost all the accuracy, if not all, is lost when performing the subtraction $x - \mathrm{RN}(kC)$. For instance, on a radix-10 computer of precision 10, if $C = \pi/2$ and $x = 8248.251512$, then $x^* = -0.0000000000021475836702\cdots$. If the range reduction is not accurate enough to make sure that numbers that small are handled accurately, the computed result may be quite different from the actual value of x^*.

The first solution consists of using multiple-precision arithmetic, but this may make the computation significantly slower. Moreover, it is not that easy to predict the accuracy with which the computation should be performed. The first method presented below is due to Cody and Waite [93, 97]. It works for rather small input arguments only but is very inexpensive. We then present a variant of that method, due to Boldo, Daumas, and Li [40], that allows to tackle significantly larger input arguments when a fused multiply-add instruction (FMA) is available. After this, we introduce an algorithm due to Kahan

[262] that allows us to find the worst cases for range reduction. Then we give two algorithms, one due to Payne and Hanek [381], another one due to Daumas et al. [126, 127] that make it possible to perform an accurate range reduction with large input arguments, without actually needing multiple-precision calculations.

11.2 Cody and Waite's Method for Range Reduction

11.2.1 The Classical Cody–Waite Reduction

The method suggested by Cody and Waite [93, 97] consists of finding two values C_1 and C_2 that are exactly representable in the floating-point system being used, and such that

- C_1 is very close to C, and is representable using a few digits[1] only (i.e., C_1 is a machine number containing the first few digits of C). A consequence is that for values of k that are not too large, kC_1 is exactly representable in the floating-point system.
- $C = C_1 + C_2$ to beyond working precision.

Then, instead of evaluating $\mathrm{RN}(x - \mathrm{RN}(kC))$, we evaluate[2]

$$\mathrm{RN}(\mathrm{RN}(x - \mathrm{RN}(kC_1)) - \mathrm{RN}(kC_2)) \tag{11.2}$$

When doing this, if k is such that kC_1 is exactly representable, the subtraction $(x - kC_1)$ is performed without any error[3], i.e., what is actually computed is

$$\mathrm{RN}(x - kC_1 - \mathrm{RN}(kC_2)).$$

Therefore (11.2) simulates a larger precision (the precision with which $C_1 + C_2$ approximates C). For instance, if $C = \pi$, typical values for C_1 and C_2, given by Cody and Waite [93] are

- in the IEEE-754 binary32 format:

$$C_1 = 201/64 = 3.140625$$
$$C_2 = 9.67653589793 \times 10^{-4};$$

- in the IEEE-754 binary64 format:

$$C_1 = 3217/1024 = 3.1416015625$$
$$C_2 = -8.90891020676153735617 \times 10^{-6}.$$

[1] In the radix of the floating-point system being used —2 in general.

[2] Gal and Bachelis [199] use the same kind of range reduction, but slightly differently. Some accuracy is lost when subtracting $\mathrm{RN}(kC_2)$ from $\mathrm{RN}(x - \mathrm{RN}(kC_1))$. To avoid this, they keep these terms separate. This is also done by Tang (see section 6.2.1).

[3] Unless subtraction is not correctly rounded. To my knowledge, there is no current computer with such a poor arithmetic.

This method helps to reduce the error due to the cancellation at a low cost. It can easily be generalized: C can be represented as the sum of three values C_1, C_2 and C_3. However, if last-bit accuracy is required, and if we want to provide a correct result for all possible input values (even if they are very large), it will be used for small arguments only.

For instance, the binary64 CRLIBM library[4] [124] (see Section 14.4) uses four possible methods for reducing the argument to trigonometric functions (with $C = \pi/256$), depending on the magnitude of the input number:

- Cody and Waite's method with two constants (the fastest);
- Cody and Waite's method with three constants (almost as fast);
- Cody and Waite's method with three constants, using a double-word arithmetic and a 64-bit integer format for the integer k;
- Payne and Hanek's algorithm (see Section 11.4) for the largest arguments.

11.2.2 When a Fused Multiply-add (FMA) Instruction is Available

In [40], Boldo, Daumas, and Li give several algorithms and properties that are of much interest when one wishes to perform range reduction with large input arguments, assuming that an FMA instruction is available. I will just present here their more accurate algorithm. Assume that we use a binary, precision-p floating-point arithmetic of minimum exponent $e_{\min}$. Let $\mathrm{RN}(\cdot)$ be the round-to-nearest (ties to even) rounding function. Define $\lceil t \rfloor$ as the integer nearest t (with any choice in case of a tie). Let C be the real constant of the additive range reduction, and assume $C > 0$. We will build a pair (C_1, C_2) of floating-point numbers whose sum is a very accurate approximation to C as follows:[5]

- define R as $\mathrm{RN}(1/C)$;
- define C_1 as
$$C_1 = \left\lceil \frac{1}{4 \cdot R \cdot \mathrm{ulp}\,(1/R)} \right\rfloor \cdot 4 \cdot \mathrm{ulp}\,(1/R),$$
(see Section 2.1.3 for a definition of the ulp function)
- and finally define C_2 as
$$C_2 = \left\lceil \frac{(C - C_1)}{2^{-p+4}\,\mathrm{ulp}\,(C_1)} \right\rfloor \cdot 2^{-p+4}\,\mathrm{ulp}\,(C_1).$$

The argument reduction itself consists in performing the following algorithm. It calls algorithm Fast2Sum (Algorithm 1, presented in Section 2.2.1). This reduction algorithm starts like a straightforward adaptation of (11.2) to the fact that an FMA instruction is available, i.e., we compute $\mathrm{RN}(\mathrm{RN}(x - kC_1) - kC_2$. Then, after this, a correction is applied. That correction is computed using an adaptation of a technique suggested by Boldo and Muller for evaluating the error of an FMA [46].

[4] http://lipforge.ens-lyon.fr/projects/crlibm/.

[5] Important notice: one can easily find an even better approximation to C as a sum of two floating-point numbers, however, it may not satisfy the property presented in Theorem 18, that is, $x - k \cdot (C_1 + C_2)$ may not be computed exactly by Algorithm 23.

Algorithm 23 *Boldo, Daumas, and Li's accurate range reduction algorithm [40]. It takes as input the initial argument x and outputs the reduced argument, represented by the double word (v_h, v_ℓ). It requires the availability of an FMA instruction.*

$k \leftarrow \mathrm{RN}(\mathrm{RN}(3 \cdot 2^{p-2} + R \cdot x) - 3 \cdot 2^{p-2})$
$u \leftarrow \mathrm{RN}(x - k \cdot C_1)$
$v_h \leftarrow \mathrm{RN}(u - k \cdot C_2)$
$\rho_h \leftarrow \mathrm{RN}(k \cdot C_2)$
$\rho_\ell \leftarrow \mathrm{RN}(k \cdot C_2 - \rho_h)$ ♯ note: $\rho_h + \rho_\ell = kC_2$ exactly
$(t_h, t_\ell) = \mathrm{Fast2Sum}(u, -\rho_h)$
$v_\ell = \mathrm{RN}(\mathrm{RN}(\mathrm{RN}(t_h - v_h) + t_\ell) - \rho_\ell$.
return (v_h, v_ℓ)

We have the following result.

Theorem 18 *With the values R, C_1, and C_2 defined above, and under the following assumptions:*

- *R is a normal floating-point number (i.e., $R \geq 2^{e_{\min}}$);*
- *C_1 is not a power of 2;*
- *$C_1 > 2^{e_{\min}+p-1}$;*
- *the input value x is such that $|R \cdot x| \leq 2^{p-2} - 1$;*

The double-word (v_h, v_ℓ) returned by Algorithm 23 satisfies

$$v_h + v_\ell = x - k \cdot (C_1 + C_2). \tag{11.3}$$

Furthermore, we have,

$$|C - (C_1 + C_2)| \leq 2^{-p+3}\, ulp\,(C_1). \tag{11.4}$$

therefore the difference between $x - k \cdot (C_1 + C_2)$ and $x - k \cdot C$ is of absolute value less than

$$k \cdot 2^{-p+3} \cdot\ ulp\,(C_1). \tag{11.5}$$

Equation (11.5) and the requirement $|R \cdot x| \leq 2^{p-2} - 1$ help to determine the largest value of $|x|$ for which the error of the range reduction will be guaranteed to be within some chosen bound ϵ. For instance, if $C = \pi$, that largest value is $(2^{2p-5}\epsilon - 1) \cdot \pi$. It is worth being noticed that many variants of Algorithm 23 are possible. For instance, if we define C_3 as $\mathrm{RN}(C - (C_1 + C_2))$ then the triple-word[6] $(v_h, v_\ell, \mathrm{RN}(-k \cdot C_3))$ is an even better approximation to the exact reduced argument.

Let us now give an example.

Example 14 (Boldo, Daumas, and Li's reduction, with $C = \pi$). *Let us assume that we use binary64 arithmetic ($p = 53$, and $e_{\min} = -1022$), and that $C = \pi$. The above-defined floating-point numbers R, C_1, and C_2 are equal to*

$$R = 5734161139222659 \cdot 2^{-54};$$
$$C1 = 884279719003555 \cdot 2^{-48};$$
$$C2 = 4851325781271 \cdot 2^{-95}.$$

[6] Beware: $RN(-k \cdot C_3)$ is not necessarily very small in front of v_ℓ. If we need that property—which is likely if we want to evaluate some approximating polynomial at the reduced argument—some further manipulations are needed.

If the input value x is equal to 103554, *when executing Algorithm 23, we successively obtain*

$$\begin{aligned} k &= 32962; \\ u &= 115820250045757 \cdot 2^{-47}; \\ v_h &= 7412496002892089 \cdot 2^{-53}; \\ \rho_h &= 4997168762570459 \cdot 2^{-90}; \\ \rho_\ell &= 7 \cdot 2^{-94}; \\ t_h &= 7412496002892089 \cdot 2^{-53}; \\ t_\ell &= -25853282011 \cdot 2^{-90}; \\ v_\ell &= -413652512183 \cdot 2^{-94}. \end{aligned}$$

The computed, double-word, reduced argument $v_h + v_\ell$ equals

$$0.822952373235273794348800346615249743 7\cdots$$

It approximates $x - k\pi$ with error $\approx 5.59 \times 10^{-27}$. This value should be compared with the error bound obtained from (11.5), *namely* 1.31×10^{-26}.

11.3 Finding Worst Cases for Range Reduction?

11.3.1 A Few Basic Notions On Continued Fractions

To estimate the potential loss of accuracy that can occur when performing range reduction, we can use a technique suggested by Kahan [262], based on the theory of continued fractions. We just recall here the elementary definitions and results on continued fractions that we need in the following. For more information, good references on continued fractions are Stark's book [430] and Khinchin's book [271].

In all this section, let α be an irrational number. The sequence of the *convergents* to α (or continued fraction approximations to α) is defined as follows. First, from α we build two sequences (a_i) and (r_i)

$$\begin{cases} r_0 &= \alpha \\ a_i &= \lfloor r_i \rfloor \\ r_{i+1} &= \dfrac{1}{r_i - a_i}. \end{cases} \tag{11.6}$$

These sequences are defined for any i (i.e., r_i is never equal to a_i; this is due to the irrationality of α), and the rational number

$$\frac{P_i}{Q_i} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_i}}}}}$$

is called the ith *convergent* to α. One can easily show that the P_is and the Q_is can be deduced from the a_is using the following recurrences,

$$\begin{aligned} P_0 &= a_0 \\ P_1 &= a_1 a_0 + 1 \\ Q_0 &= 1 \\ Q_1 &= a_1 \\ P_n &= P_{n-1} a_n + P_{n-2} \\ Q_n &= Q_{n-1} a_n + Q_{n-2}. \end{aligned}$$

The main interest of continued fractions lies in the following theorem.

Theorem 19 *For all irrational α, if the convergents to α are the terms P_i/Q_i and if $i \geq 2$, then for any rational number p/q, that satisfies $p/q \neq P_i/Q_i$, $p/q \neq P_{i+1}/Q_{i+1}$, and $q \leq Q_{i+1}$, we have*

$$\left|\alpha - \frac{P_i}{Q_i}\right| < \frac{q}{Q_i}\left|\alpha - \frac{p}{q}\right|.$$

A consequence of this is that if a rational number p/q approximates α better than P_i/Q_i, then $q > Q_i$.

In other words, P_i/Q_i is the best approximation to α among the rational numbers whose denominator is less than or equal to Q_i.

Moreover, one can show that

$$\left|\alpha - \frac{P_i}{Q_i}\right| < \frac{1}{Q_i Q_{i+1}}.$$

We write:

$$\alpha = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}}$$

For instance

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{\ddots}}}}},$$

which gives the following rational approximations to π,

$$\frac{P_0}{Q_0} = 3 \quad \frac{P_1}{Q_1} = \frac{22}{7} \quad \frac{P_2}{Q_2} = \frac{333}{106} \quad \frac{P_3}{Q_3} = \frac{355}{113} \quad \frac{P_4}{Q_4} = \frac{103993}{33102}$$

and

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \ddots}}}}}$$

Beyond our current problem of finding worst cases for range reduction, continued fractions are of interest in many areas of computer arithmetic, such as rational arithmetic [281, 420], exact real arithmetic [467], correctly-rounded fast multiplication by an arbitrary-precision constant in floating-point arithmetic [67], or—as we will soon see—test to determine if the sine or cosine of a normal FP number can underflow.

11.3.2 Finding Worst Cases Using Continued Fractions

Now let us turn to the application of continued fractions to range reduction. We assume we use a radix-β floating-point system of precision p, and exponents between $e_{\min}$ and $e_{\max}$. We also assume that we want to perform an additive range reduction, and that the constant C is an irrational number (this is always the case, since in practice, C is π times a power of two—for trigonometric functions—or a simple rational number times the logarithm of the radix of the number system being used—for exponentials). Suppose that an input number x is the floating-point number:

$$x = x_0.x_1x_2x_3\cdots x_{p-1} \times \beta^{exponent}, \text{ with } x_0 \neq 0.$$

We can rewrite x with an integral significand:

$$x = M \times \beta^{exponent-p+1},$$

where $M = x_0x_1x_2x_3\cdots x_{p-1}$ satisfies $\beta^{p-1} \leq M \leq \beta^p - 1$. Performing the range reduction is equivalent to finding an integer k and a real number s, $|s| < 1$ such that

$$\frac{x}{C} = k + s. \tag{11.7}$$

The number x^* of the introduction of this chapter is equal to sC. We can rewrite (11.7) as

$$\frac{\beta^{exponent-p+1}}{C} = \frac{k}{M} + \frac{s}{M}.$$

If the result x^* of the range reduction is a very small number ϵ, this means that

$$\frac{\beta^{exponent-p+1}}{C} = \frac{k}{M} + \frac{\epsilon}{MC} \approx \frac{k}{M}. \tag{11.8}$$

In such a case, k/M must be a *very good rational approximation* to the irrational number $\beta^{exponent-p+1}/C$. To estimate the smallest possible value of ϵ, it suffices to examine the sequence of the "best" possible rational approximations to $\beta^{exponent-p+1}/C$, that is, the sequence of its con-

vergents. We proceed as follows: for each considered value of *exponent* (of course, the exponents corresponding to a value of x smaller than C do not need to be considered), we compute the first terms P_i and Q_i of the convergents to $\beta^{exponent-p+1}/C$. We then select the approximation P_j/Q_j whose denominator is the largest less than β^p. From Theorem 19, we know that for any rational number P/M, with $M \leq \beta^p - 1$, we have

$$\frac{M}{Q_j}\left|\frac{P}{M} - \frac{\beta^{exponent-p+1}}{C}\right| \geq \left|\frac{P_j}{Q_j} - \frac{\beta^{exponent-p+1}}{C}\right|.$$

Therefore,

$$\left|\frac{P}{M} - \frac{\beta^{exponent-p+1}}{C}\right| \geq \frac{Q_j}{M}\left|\frac{P_j}{Q_j} - \frac{\beta^{exponent-p+1}}{C}\right|.$$

Therefore,

$$MC\left|\frac{P}{M} - \frac{\beta^{exponent-p+1}}{C}\right| \geq CQ_j\left|\frac{P_j}{Q_j} - \frac{\beta^{exponent-p+1}}{C}\right|.$$

Hence, the lowest possible value[7] of ϵ is attained for $M = Q_j$ and $P = P_j$, and is equal to

$$CQ_j\left|\frac{P_j}{Q_j} - \frac{\beta^{exponent-p+1}}{C}\right|.$$

To find the worst case for range reduction, it suffices to compute P_j and Q_j for each value of the exponent that allows the representation of numbers greater than C. This method was first suggested by Kahan [262] (a C-program that implements this method without needing extra-precision arithmetic was written by Kahan and McDonald. As I am writing this book, it is accessible at http://www.eecs.berkeley.edu/~wkahan/testpi/nearpi.c). A similar method was found later by Smith [428]. The following Maple program implements this method:

```
worstcaseRR := proc(B,p,emin,emax,C,ndigits)
  local epsilonmin,powerofBoverC,e,a,Plast,r,Qlast,
        Q,P,NewQ,NewP,epsilon,
      numbermin,expmin,ell;
      epsilonmin := 12345.0 ;
      Digits := ndigits;
      powerofBoverC := B^(emin-p)/C;
      for e from emin-p+1 to emax-p+1 do
          powerofBoverC := B*powerofBoverC;
          a := floor(powerofBoverC);
```

[7] If Q_j is less than β^{p-1}, this value does not actually correspond to a floating-point number of exponent *exponent*. In such a case, the actual lowest value of ϵ is larger. However, the value corresponding to Q_j is actually attained for another value of the exponent: let ℓ be the integer such that $\beta^{p-1} \leq \beta^\ell Q_j < \beta^p$; from

$$CQ_j\left|\frac{P_j}{Q_j} - \frac{\beta^{exponent-p+1}}{C}\right| = C\beta^\ell Q_j\left|\frac{P_j}{\beta^\ell Q_j} - \frac{\beta^{exponent-\ell-p+1}}{C}\right|$$

we deduce that the value of ϵ corresponding to Q_j is actually attained when the exponent equals $exponent - \ell$. Therefore the fact that Q_j may be less than β^{p-1} has no influence on the final result, that is, the search for the lowest possible value of ϵ.

```
            Plast := a;
            r := 1/(powerofBoverC-a);
            a := floor(r);
            Qlast := 1;
            Q := a;
            P := Plast*a+1;
            while Q < B^p-1 do
                r := 1/(r-a);
                a := floor(r);
                NewQ := Q*a+Qlast;
                NewP := P*a+Plast;
                Qlast := Q;
                Plast := P;
                Q := NewQ;
                P := NewP
            od;
            epsilon :=
                evalf(C*abs(Plast-Qlast*powerofBoverC));
              if epsilon < epsilonmin then
                epsilonmin := epsilon; numbermin := Qlast;
                expmin := e
            fi
        od;
        print('significand',numbermin);
        print('exponent',expmin);
        print('epsilon',epsilonmin);
        ell := evalf(log(epsilonmin)/log(B),10);
        print('numberofdigits',ell)
    end
```

Various results obtained using this program are given in Table 11.2. These results can be used to perform range reduction accurately using one of the algorithms given in the next sections, or simply to check whether the range reduction is accurately performed in a given package.

In Table 11.2, the number $-\log_\beta(\epsilon)$ makes it possible to deduce the precision with which the computations must be carried out to get the reduced argument with a given accuracy. Assume that we want to get the reduced argument x^* with, say, m significant radix-β digits. Since x^* may be as small as ϵ, it must be computed with an absolute error less than $\beta^{-m-\log_\beta(\epsilon)}$. Roughly speaking, $-\log_\beta(\epsilon)$ is the number of "guard digits" that are necessary to perform the range reduction accurately. By examining Table 11.2, we can see that $-\log_\beta(\epsilon)$ is always close to the number of digits of the number system (i.e., p plus the number of exponent digits). This is not surprising: if we assume that the digits of the fractional parts of the reduced arguments are independent random variables with probability $1/\beta$ for each possible digit, then we can show that the most probable value for $-\log_\beta(\epsilon)$ is close to p plus the radix-β logarithm of the number of exponent digits. A similar probabilistic argument is used when examining the possibility of correctly rounding the elementary functions (see Section 12.6).

Table 11.2 *Worst cases for range reduction for various floating-point systems and reduction constants C.*

β	p	C	$e_{\max}$	Worst Case	$-\log_\beta(\epsilon)$
2	24	$\pi/2$	127	$16367173 \times 2^{+72}$	29.2
2	24	$\pi/4$	127	$16367173 \times 2^{+71}$	30.2
2	24	ln(2)	127	8885060×2^{-11}	31.6
2	24	ln(10)	127	9054133×2^{-18}	28.4
10	10	$\pi/2$	99	$8248251512 \times 10^{-6}$	11.7
10	10	$\pi/4$	99	$4124125756 \times 10^{-6}$	11.9
10	10	ln(10)	99	$7908257897 \times 10^{+30}$	11.7
2	53	$\pi/2$	1023	$6381956970095103 \times 2^{+797}$	60.9
2	53	$\pi/4$	1023	$6381956970095103 \times 2^{+796}$	61.9
2	53	ln(2)	1023	$5261692873635770 \times 2^{+499}$	66.8
2	113	$\pi/2$	1024	$614799\cdots1953734 \times 2^{+797}$	122.79

11.4 The Payne and Hanek Reduction Algorithm

Payne and Hanek [381] designed a range reduction algorithm for the trigonometric functions that is very interesting when the input argument is very large. Assume we wish to compute

$$y = x - kC, \tag{11.9}$$

where $C = 2^{-t}\pi$ is the constant of the range reduction. Assume x is a binary floating-point number of precision p, and define e_x as its exponent and X as its integral significand, so that $x = X \times 2^{e_x - p + 1}$, where X is an integer, $2^{p-1} \leq |X| \leq 2^p - 1$. Equation (11.9) can be rewritten as

$$y = 2^{-t}\pi \left(\frac{2^{t+e_x-p+1}}{\pi} X - k \right). \tag{11.10}$$

Let us denote

$$0.v_1v_2v_3v_4v_5v_6v_7\cdots$$

the infinite binary expansion of $1/\pi$. The beginning of that expansion is

$$0.01010001011111001100000110110111001001110010001000001010100101001111111$$
$$00001001110101011111010001111101010011010011011101110000001101101101100 0$$
$$10100101011001100100111100010000111001000001000001111111100101000101 1\cdots$$

The main idea behind Payne and Hanek's algorithm is to note that when computing y using (11.10), we do not need to use too many bits of $1/\pi$ for evaluating the product $\frac{2^{t+e_x-p+1}}{\pi} X$:

- the bit v_i of $1/\pi$ is of weight 2^{-i}. Once multiplied by $2^{t+e_x-p+1}X$, the term $v_i 2^{-i}$ will become a multiple of $2^{t+e_x-p+1-i}$. Hence, once multiplied, later on, by $2^{-t}\pi$, it will become a multiple of $2^{e_x-p+1-i}\pi$. Therefore, as soon as $i \leq e_x - p$, the contribution of bit v_i in Equation (11.10) will result in a multiple of 2π: it will have no influence on the trigonometric functions. So, in the product

$$\frac{2^{t+e_x-p+1}}{\pi} X,$$

we can replace the bits of $1/\pi$ of rank i less than or equal to $e_x - p$ by zeros;

- since $|X| \leq 2^p - 1$, the contribution of all the bits $v_i, v_{i+1}, v_{i+2}, v_{i+3}, \ldots$ in the reduced argument is less than
$$2^{-t}\pi \times 2^{t+e_x-p+1} \times 2^{-i+1} \times 2^p < 2^{e_x-i+4};$$
therefore, if we want the reduced argument with an absolute error less than $2^{-\ell}$, we can replace the bits of $1/\pi$ of rank i larger than or equal to $4 + \ell + e_x$ by zeros.

Therefore, Payne and Hanek's reduction consists in performing the computation (11.10) with only bits of rank $e_x - p + 1$ to $e_x + 3 + \ell$ of $1/\pi$. This requires a product of a p-bit number (the number $X \cdot 2^{t+e_x-p+1}$) by a $3 + p + \ell$-bit number. Such a product is easily computed. The last multiplication by π in 11.10) does not require more than $\ell - t + 2$ bits of π.

Example 15 (Payne and Hanek's algorithm). *Assume that we want to evaluate the sine of* $x = 1.1_2 \times 2^{200}$, *that the constant of the range reduction is* $\pi/2$ (*i.e.,* $t = 1$), *and that we use an IEEE-754 binary64 arithmetic. Also assume that we choose* $\ell = 20$. *We have,*

$$\begin{aligned} p &= 53 \\ e_x &= 200 \\ X &= x \times 2^{p-1-e} = 6755399441055744 \end{aligned}$$

Only the digits of rank $e_x - p + 1 = 148$ *to* $e_x + 3 + \ell = 223$ *of* $1/\pi$ *need to be used in the multiplication. That is, in* (11.10), *we can replace*

$$\frac{2^{t+e_x-p+1}}{\pi}$$

by

$$G = \frac{12749548807077367524469}{2^{73}}.$$

The product $G \cdot X$ *equals*:

$$\frac{3824864642123210257340 7}{2^{22}}$$

$$= 9119187932308221 + \frac{2000223}{2^{22}},$$

therefore, we choose $k = \lfloor G \cdot X \rfloor = 9119187932308221$, *and if we use the best binary64 approximation to* π, say $\hat{\pi}$, *for performing the final product, the number*

$$y = 2^{-t}\hat{\pi}\,(G \cdot X - k) \tag{11.11}$$

is equal to $0.7490975716520949311090\cdots$, *whereas the "exact" reduced argument is* $0.7490981151628151254101545\cdots$. *One easily checks that this corresponds to an accuracy of slightly more than* 20 *bits, as required.*

11.5 Modular Range Reduction Algorithms

Now, we assume that we have an algorithm able to compute the function g of the introduction of this chapter in an interval I of the form $[-C/2-\epsilon, +C/2+\epsilon]$ (we call this case *"symmetrical reduction"*) or $[-\epsilon, C+\epsilon]$ (we call this case *"positive reduction"*), with $\epsilon \geq 0$. We still want to find $x^* \in I$ and an integer k such that

$$x^* = x - kC. \tag{11.12}$$

If $\epsilon > 0$, then x^* and k are not uniquely defined by Eq. 11.12. In such a case, the problem of deducing these values from x is called *"redundant range reduction"*. For example, if $C = \pi/2$, $I = [-1, 1]$, and $x = 2.5$, then $k = 1$ and $x^* = 0.929203\cdots$ or $k = 2$ and $x^* = -0.641592\cdots$ are possible values. As in many fields of computer arithmetic, redundancy will allow faster algorithms. It is worth noticing that with all usual algorithms for evaluating the elementary functions, one can assume that the length of the convergence domain I is greater than C, i.e., that we can perform a *redundant* range reduction. For instance, with the CORDIC algorithm, when performing rotations (see Chapter 9), the convergence domain is $[-1.743\ldots, +1.743\ldots]$, which is much larger than $[-\pi/2, +\pi/2]$. With polynomial or rational approximations, the convergence domain can be enlarged if needed by adding one coefficient to the approximation.

We first present here an algorithm proposed by Daumas et al. [126, 127]. This algorithm is called the *modular range reduction algorithm* (MRR). Then, we present variants and improvements due to Villalba et al. [464], and to Brisebarre et al. [64].

11.5.1 The MRR Algorithm

11.5.1.1 MRR: Fixed-Point Reduction

First of all, we assume that the input operands are *fixed-point radix-2 numbers*, less than 2^N. These numbers have an N-bit integer part and a p-bit fractional part. So, the digit chain:

$$x_{N-1}x_{N-2}x_{N-3}\ldots x_0.x_{-1}x_{-2}\cdots x_{-p}, \text{ where } x_i \in \{0, 1\}$$

represents the number

$$\sum_{i=-p}^{N-1} x_i 2^i.$$

We assume that we should perform a *redundant* range reduction, and we call v the integer such that $2^v < C \leq 2^{v+1}$.

Let us define, for $i \geq v$, the number $m_i \in [-C/2, C/2)$ such that $(2^i - m_i)/C$ is an integer (in the following, we write "$m_i \equiv 2^i \bmod C$"). The Modular Range Reduction (MRR) algorithm consists of performing the following steps.

1. First reduction We compute the number[8]:

$$r = (x_{N-1}m_{N-1}) + (x_{N-2}m_{N-2}) + \ldots + (x_\nu m_\nu) \\ +x_{\nu-1}x_{\nu-2}x_{\nu-3}\ldots x_0.x_{-1}x_{-2}\cdots x_{-p}. \tag{11.13}$$

Since the x_is are equal to 0 or 1, this computation is reduced to the sum of at most $N-\nu+1$ terms. The result r of this first reduction is between $-(N-\nu+2)C/2$ and $+(N-\nu+2)C/2$. This is a consequence of the fact that all the $x_i m_i$ have an absolute value smaller than $C/2$, and

$$x_{\nu-1}x_{\nu-2}x_{\nu-3}\cdots x_0.x_{-1}x_{-2}\cdots x_{-p}$$

has an absolute value less than 2^ν, which is less than C.

2. Second reduction We define the r_is as the digits of the result of the first reduction:

$$r = r_\ell r_{\ell-1}r_{\ell-2}\cdots r_0.r_{-1}r_{-2}\cdots,$$

where $\ell = \lfloor \log_2((N-\nu+2)C/2)\rfloor$.
We also define $\hat{r}$ as the number obtained by truncating the binary representation of r after the $\lceil -\log_2(\epsilon)\rceil$th fractional bit, that is:

$$\hat{r} = r_\ell r_{\ell-1}r_{\ell-2}\cdots r_0.r_{-1}r_{-2}\cdots r_{\lfloor \log_2(\epsilon)\rfloor};$$

$\hat{r}$ is an m-digit number, where the number

$$m = \lfloor \log_2((N-\nu+2)C/2)\rfloor + \lceil -\log_2(\epsilon)\rceil$$

is very small in all practical cases (see the following example). If we define k as[9] $\lfloor \hat{r}/C\rceil$ (resp., $\lfloor \hat{r}/C\rfloor$) then $r - kC$ will belong to $[-C/2-\epsilon, +C/2+\epsilon]$ (resp., $[-\epsilon, C+\epsilon]$); that is, it will be the correct result of the symmetrical (resp., positive) range reduction.

Proof

1. *In the symmetrical case.* We have $|k - \hat{r}/C| \le 1/2$; therefore $|\hat{r} - kC| \le C/2$. From the definition of $\hat{r}$, we have
$$|r - \hat{r}| \le 2^{\lfloor \log_2(\epsilon)\rfloor} \le \epsilon;$$
therefore:
$$|r - kC| \le \frac{C}{2} + \epsilon.$$

2. *In the positive case.* We have $k \le \hat{r}/C < k+1$; therefore $0 \le \hat{r} - kC < C$; therefore $-\epsilon \le r - kC < C + \epsilon$.

[8] This formula looks correct only for positive values of ν. It would be more correct, although maybe less clear, to write: $r = \sum_{i=\nu}^{N-1} x_i m_i + \sum_{i=-p}^{\nu-1} x_i 2^i$.

[9] We denote $\lfloor x\rceil$ the integer that is nearest to x.

Since k can be deduced from $\hat{r}$, this second reduction step will be implemented by looking up the value kC in a 2^m-bit entry table at the address constituted by the bits of $\hat{r}$.

During this reduction process, we perform the addition of $N - \nu + 1$ terms. If these terms (namely, the m_is and the value kC of the second reduction step) are represented in fixed-point with q fractional bits (i.e., the error on each of these terms is bounded by 2^{-q-1}), then the difference between the result of the computation and the exact reduced argument is bounded by $2^{-q-1}(N-\nu+1)$. In order to obtain the reduced argument x^* with the same absolute accuracy as the input argument x (i.e., p significant fixed-point fractional digits), one needs to store the m_is and the values kC with $p + \lceil \log_2(N - \nu + 1) \rceil$ fractional bits.

Example 16 *Assume we need to compute sines of angles between* -2^{20} *and* 2^{20}, *and that the algorithm used with the reduced arguments is CORDIC (see Chapter 9). The convergence interval* I *is* $[-1.743, \cdots, +1.743\cdots]$; *therefore (since* $1.743 > \pi/2$*) we have to perform a* symmetrical redundant *range reduction, with* $C = \pi$ and $\epsilon = +1.743\cdots - \pi/2 = 0.172\cdots > 2^{-3}$. *We immediately get the following parameters*.

- $N = 20$ and $\nu = 2$.
- *The first range reduction consists of the addition of 19 terms*.
- $r \in [-10\pi, +10\pi]$; *therefore, since* $10\pi < 2^5$, *the second reduction step requires a table with* $5 + \lceil -\log_2 \epsilon \rceil = 8$ *address bits*.
- *To obtain the reduced argument with* p *significant fractional bits, one needs to store the* m_is *and the values* kC *with* $p + 5$ bits.

Assume we compute $\sin(355)$. *The binary representation of* 355 is 101100011. *Therefore during the first reduction, we have to compute* $m_8 + m_6 + m_5 + m_1 + 1$, *where*:

- $m_8 = 256 - 81 \times \pi = 1.530995059226747684\cdots$
- $m_6 = 64 - 20 \times \pi = 1.1681469282041352307\cdots$
- $m_5 = 32 - 10 \times \pi = 0.5840734641020676153\cdots$
- $m_1 = 2 - \pi = -1.141592653589793238462\cdots$.

We get $m_8 + m_6 + m_5 + m_1 + 1 = 3.1416227979431572921\cdots$. *The second reduction consists in subtracting* π *from that result, which gives*

$$0.00003014435336405372\cdots,$$

the sine of which is $0.000030144353359488449\cdots$. *Therefore*

$$\sin(355) = -0.000030144353359488449\cdots.$$

11.5.1.2 MRR: Floating-Point Reduction

Now assume that the input value x is a radix-2 floating-point number:

$$x = x_0.x_1x_2x_3\cdots x_{n-1} \times 2^{exponent}.$$

The range reduction can be performed exactly as in the fixed-point case. During the first reduction, we replace the addition of the terms m_i by the addition of the terms $m_{exponent-i} \equiv 2^{exponent-i} \bmod C$. During the reduction process, we just add numbers (the m_is) of the same order of magnitude, represented in fixed-point. This helps to make the reduction accurate. One can easily show that if m_is and the terms

kC of the second reduction are represented with q fractional bits, then the *absolute* error on the reduced argument is bounded by $(n+1)2^{-q-1}$. If we want a reduced argument with at least t significant bits, and if the number $-\log_2(\epsilon)$ found using the algorithm presented in Section 11.3.2 is less than some integer j, then $q = j + t - 1 + \lceil \log_2(n+1) \rceil$ is convenient.

11.5.1.3 Architectures for the MRR algorithm

The first reduction consists of adding $N - \nu + 1$ numbers. This addition can be performed in a redundant number system in order to benefit from the carry-free ability of such a system, and/or with a tree of adders. This problem is obviously closely related to the problem of multiplying two numbers (multiplying $x = \sum_{i=0}^{q} x_i 2^i$ by $y = \sum_{j=0}^{q} y_i 2^j$ reduces to computing the sum of the $q+1$ terms $y_j 2^j x$). Therefore, almost all the classical architectures proposed in the multiplication literature (see for instance [53, 121, 215, 358, 446, 468]), can be slightly modified in order to be used for range reduction. This similarity between modular range reduction and multiplication makes it possible to perform both operations with the same hardware, which can save some silicon area on a circuit. To accelerate the first reduction, we can perform a Booth recoding [47], or merely a modified Booth recoding [244], of x. This would give a signed digit (with digits -1, 0, and 1) representation of x with at least half of the digits equal to zero. Then, the number of terms to be added during the first reduction would be halved.

A modified version of the MRR algorithm that works "on the fly" was introduced by Lefèvre and Muller [309].

11.5.2 The Double Residue Modular Range Reduction (DRMRR) Algorithm

The major drawback of the MRR algorithm is the need to perform a second reduction step. Villalba, Lang, and Gonzalez [464] find an ingenious way of solving that problem, inspired from digit-recurrence division algorithms. Let us assume that we use fixed-point arithmetic (the floating-point case is not so different), and that C is the constant of the range reduction. Their *double residue modular range reduction* (DRMRR) algorithm is as follows. We start from the initial argument

$$x_{N-1}x_{N-2}x_{N-3}\ldots x_0.x_{-1}x_{-2}\cdots x_{-p}, \text{ where } x_i \in \{0,1\}.$$

Instead of adding the constants $m_i = x_i \cdot (2^i \mod C)$, Villalba, Lang, and Gonzalez consider two different constants for each value of i:

- m_i^+ is the number of the form $2^i - k \cdot C$ such that[10] $0 \le m_i^+ < C$ and k is an integer;
- m_i^- is the number of the form $2^i - k \cdot C$ such that $-C \le m_i^- < 0$ and k is an integer.

At step i of the range reduction algorithm, the decision to use m_{N-i}^+ or m_{N-i}^- is taken by examining the sign of the sum R_{i-1} of the already accumulated terms. More precisely, we start from $R_0 = 0$, and for $i = 1, \ldots, N+p$, we iterate

$$R_i = \begin{cases} R_{i-1} + x_{N-i} \cdot m_{N-i}^- & \text{if } R_{i-1} \ge 0, \\ R_{i-1} + x_{N-i} \cdot m_{N-i}^+ & \text{if } R_{i-1} < 0, \end{cases} \tag{11.14}$$

[10] In all practical cases, C is a transcendental number, so that m_i^+ and m_i^- are never equal to $-C$, 0, or $+C$.

The reduced argument is R_{N+p}. One easily checks that it is between $-C$ and $+C$. Villalba, Lang, and Gonzalez also suggest a *redundant* variant of their algorithm that is of much interest for a hardware implementation: assuming that the terms R_i are represented in redundant (say, carry-save or borrow-save) representation, we make a decision similar to (11.14) on the basis of a small window of digits of R_{i-1} only. See [464] for more details. A variant of their algorithm more suitable for pipelined computation is proposed in [251]. The DRMRR algorithm is compared to various range reduction algorithms in [251]. A circuit implementation of a variant of DRMRR is presented in [390].

11.5.3 High Radix Modular Reduction

Brisebarre, Defour, Kornerup, Muller, and Revol [64, 147] present an algorithm for dealing with arguments of "reasonable size" (for small arguments, variants of Cody and Waite's algorithm are very efficient, and for very large arguments they suggest to use Payne and Hanek's algorithm). Their method can be viewed as a high radix modular algorithm: the input binary64 argument is split into eight 8-bit parts, and each part is used to address tables of binary64 numbers. See [64] for more details.

Final Rounding

12

12.1 Introduction

This chapter is devoted to the problems of preserving monotonicity and always getting correctly rounded results when implementing the elementary functions in floating-point arithmetic.

Preserving monotonicity is important. That is, if a function f is monotonically increasing (resp., decreasing) over an interval, then its computer approximation must also be monotonically increasing (resp., decreasing). As pointed out by Silverstein et al. [425], monotonicity failures can cause severe problems, for example, in evaluating divided differences.

Requiring correctly rounded results not only improves the accuracy of computations: it is the only way to make numerical software portable. Portability would need a standardization of the elementary functions, and a standard cannot be widely accepted if some implementations are better than the standard. Moreover, as noticed by Agarwal et al. [2], correct rounding facilitates the preservation of useful mathematical properties such as monotonicity,[1] symmetry,[2] and important identities. And yet, in some very rare cases, correct rounding may prevent satisfying the range limits requirement [2] (see Chapter 1). For instance, assume that we use an IEEE-754 single-precision/binary32 arithmetic. The machine number which is closest to $\pi/2$ is

$$\ell = \frac{13176795}{8388608} = 1.5707963705062866210937 5 > \frac{\pi}{2}.$$

If the arctangent function is implemented with the round-to-nearest (ties to even or to away) rounding function, then the computed value of the arctangent of any number greater than or equal to 62919776 will be ℓ, and therefore will be larger than $\pi/2$. A consequence of that (if the tangent is correctly rounded too) is that for any single-precision number $x \geq 62919776$, the computed value of $\tan(\arctan(x))$, namely

$$\mathrm{RN}\left[\tan\left(\mathrm{RN}\left(\arctan(x)\right)\right)\right]$$

will be equal to

$$-22877332 = \tan(\ell).$$

[1] Rounding functions are increasing functions, therefore, for any rounding function $\circ(\cdot)$, if the "exact function" f is monotonic, and if correct rounding is provided, then the "computed function" f—equal to $\circ(f)$ is monotonic too.

[2] Correct rounding preserves symmetry if we round to the nearest or toward zero, that is, if the rounding function itself is symmetrical.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_12

Reference [146] presents some suggestions on the standardization of elementary functions in floating-point arithmetic. The 1985 version of the IEEE 754 Standard for Floating-Point Arithmetic did not specify anything concerning the elementary function. This was because it has been believed for years that correctly rounded functions would be much too slow at least for some input arguments. The situation changed since then and the 2008 version of the standard recommends (yet does not require) that some functions be correctly rounded:

$$\begin{gathered} e^x, e^x - 1, 2^x, 2^x - 1, 10^x, 10^x - 1, \\ \ln(x), \log_2(x), \log_{10}(x), \ln(1+x), \log_2(1+x), \log_{10}(1+x), \\ \sqrt{x^2+y^2}, 1/\sqrt{x}, (1+x)^n, x^n, x^{1/n} (n \text{ is an integer}), x^y, \\ \sin(\pi x), \cos(\pi x), \arctan(x)/\pi, \arctan(y/x)/\pi, \\ \sin(x), \cos(x), \tan(x), \arcsin(x), \arccos(x), \arctan(x), \arctan(y/x), \\ \sinh(x), \cosh(x), \tanh(x), \sinh^{-1}(x), \cosh^{-1}(x), \tanh^{-1}(x). \end{gathered}$$

In this chapter, we want to implement a function f in a radix-2 floating-point number system of precision p (i.e., with p-bit significands), and exponents between $e_{\min}$ and $e_{\max}$. Let x be a machine number. We assume that we first compute an approximation $F(x)$ of $f(x)$ with extra accuracy, so that an error bounded by some ϵ is committed. After that, this intermediate result is rounded to the precision-p target format, according to the active rounding function. Our goal is to estimate what value of ϵ must be chosen to make sure that the implementation satisfies the monotonicity criterion, or that the results are always equal to what would be obtained if $f(x)$ were first computed *exactly*, and then rounded.

12.2 Monotonicity

Without loss of generality, let us assume that the function f to be computed is increasing in the considered domain, and that its approximation[3] F is such that an *absolute error* bounded by ϵ is made. Let x be a machine number. We have

$$f\left(x + \text{ulp}(x)\right) > f(x).$$

The computed value $F\left(x + \text{ulp}(x)\right)$ is larger than or equal to $f\left(x + \text{ulp}(x)\right) - \epsilon$, and $F(x)$ is less than or equal to $f(x) + \epsilon$. Therefore, to make sure that

$$F\left(x + \text{ulp}(x)\right) \geq F(x)$$

it is sufficient that $f\left(x + \text{ulp}(x)\right) - f(x) \geq 2\epsilon$. There exists a number $\xi \in \left[x, x + \text{ulp}(x)\right]$ such that

$$f\left(x + \text{ulp}(x)\right) - f(x) = \text{ulp}(x) \times f'(\xi).$$

Therefore if

$$\epsilon \leq \frac{1}{2}\text{ulp}(x) \times \min_{t \in [x, x+\text{ulp}(x)]} \left|f'(t)\right|, \tag{12.1}$$

then the obtained implementation of f will be monotonic. Let us examine an example.

[3] Computed with a precision somewhat larger than the "target precision.".

Example 17 (Monotonic evaluation of sines) *Assume that we want to evaluate the sine function on* $[-\pi/4, +\pi/4]$. *If* $f(x) = \sin(x)$, *then*

$$\min_{[-\pi/4,+\pi/4]} \left|f'\right| = \cos\frac{\pi}{4} = \frac{\sqrt{2}}{2}.$$

Therefore if

$$\epsilon \leq \frac{\sqrt{2}}{4} \times \text{ulp}(x),$$

then condition (12.1) *is satisfied. For* $x \in [-\pi/4, +\pi/4]$, *we obviously have*

$$\frac{\text{ulp}(\sin(x))}{\text{ulp}(x)} \in \left\{\frac{1}{2}, 1\right\}.$$

Therefore if

$$\frac{\epsilon}{\text{ulp}(\sin(x))} \leq \frac{\sqrt{2}}{4},$$

then condition (12.1) *is satisfied. This means that if the error of the approximation is less than* 0.354 *ulps of the target format, then the approximation is monotonic. Roughly speaking, an approximation of the sine function on* $[-\pi/4, +\pi/4]$ *that is accurate to* $p + 2$ *bits of precision will necessarily be monotonic when rounded to* p *bits of precision [188].*

Ferguson and Brightman [188] gave techniques that can be used to prove that an approximation is monotonic. Their main result is the following theorem.

Theorem 20 (Ferguson and Brightman). *Let* $f(x)$ *be a monotonic function defined on the interval* $[a, b]$. *Let* $F(x)$ *be an approximation of* $f(x)$ *whose associated relative error is less than or equal to* ϵ, $\epsilon < 1$. *If for every pair* $m < m^+$ *of consecutive machine numbers in* $[a, b]$

$$\epsilon < \frac{|f(m^+) - f(m)|}{|f(m^+)| + |f(m)|},$$

then $F(x)$ *exhibits on the set of machine numbers in* $[a, b]$ *the same monotonic behavior exhibited by* $f(x)$ *on* $[a, b]$.

12.3 Correct Rounding: Presentation of the Problem

We assume that from any real number x and any integer m (with $m > p$), we are able to compute an approximation to $f(x)$ with an error in its significand y less than or equal to 2^{-m}. The computation can be carried out using a larger fixed-point or floating-point format, for instance with one of the algorithms presented in the previous chapters this book.

Therefore the problem is to get a precision-p floating-point correctly rounded result from the significand y of an approximation of $f(x)$, with error $\pm 2^{-m}$. One can easily see that this is not possible if y has the form:

- in rounding to the nearest mode,

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}1000000\cdots000000}^{m \text{ bits}}xxx\cdots$$

or

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}0111111\cdots111111}^{m \text{ bits}}xxx\cdots;$$

- in rounding toward $+\infty$, or toward $-\infty$ modes,

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}0000000\cdots000000}^{m \text{ bits}}xxx\cdots$$

or

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}1111111\cdots111111}^{m \text{ bits}}xxx\cdots.$$

This problem is known as the *Table Maker's Dilemma* [206]. The name *Table Maker's Dilemma* (TMD) was coined by Kahan [267].

Let us denote $\diamond$ the rounding function. We will call a *breakpoint* a value z where the rounding function changes, that is, if t_1 and t_2 are real numbers satisfying $t_1 < z < t_2$, then $\diamond(t_1) < \diamond(t_2)$.

For the "directed" rounding functions (i.e., toward $+\infty$, $-\infty$ or 0), the breakpoints are the floating-point numbers. For the rounding to the nearest rounding functions, they are the exact middle of two consecutive floating-point numbers. The TMD occurs for function f at point x (x is a floating-point number) if $f(x)$ is very close to a breakpoint.

For example, assuming a floating-point arithmetic with 6-bit significands,

$$\sin(11.1010) = 0.0\boxed{111011}01111110\cdots,$$

a problem may occur with rounding to the nearest if the sine function is not computed accurately enough.

The worst case for the natural logarithm in the full IEEE-754 binary64/double-precision range [308] is attained for

$$x = 1.0110001010101000100001100001001101100010100110110110 \times 2^{678}$$

whose logarithm is

$$\log x = \overbrace{111010110.01000111100111101\cdots110001}^{53 \text{ bits}}\underbrace{000000000000000000\cdots000000000000000}_{65 \text{ zeros}}1110...$$

This is a "difficult case" with a *directed* rounding function since it is very near a floating-point number. One of the two worst cases for radix-2 exponentials in the full double-precision range [08] is

$$1.11100100010110010110010100100110101111111$$
$$100101001101 \times 2^{-10}$$

whose radix-2 exponential is

$$\overbrace{1.00000000010100111111110000101110\cdots0011}^{53 \text{ bits}}$$
$$0\underbrace{1111111111111111\cdots1111111111111111}_{59 \text{ ones}}0100\cdots$$

It is a difficult case for *rounding to the nearest,* since it is very close to the middle of two consecutive floating-point numbers.

Ziv's "multilevel strategy" [481], illustrated in Figure 12.1, consists of starting to approximate the function with relative error $\pm 2^{-m_0}$, where m_0 is small but larger than p (say, $m_0 \approx p+10$ or $p+20$). In most cases,[4] that approximation will suffice to get the correctly rounded result. If it does not suffice, then another attempt is made with a significantly larger value of m, say m_1. Again, in most cases,

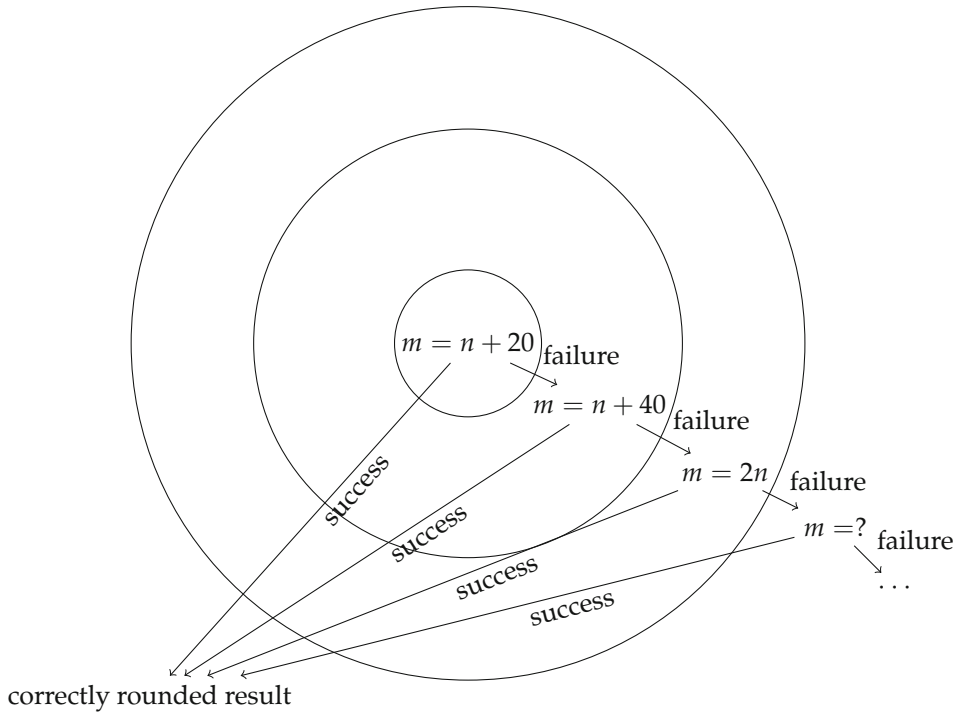


Figure 12.1 *Ziv's multilevel strategy.*

[4]The probability of a failure is about one over one million with $m_0 = p + 20$.

the new approximation will suffice. If it does not, further approximations are computed with larger and larger values of m. This strategy was implemented in the LIBULTIM library (see Section 14.3). Of course, when m increases, computing the approximations requires more and more time, but the probability that a very large m is needed is so small that in practice, the average computation time is only slightly larger than the time required with $m = m_0$. Our problem is to know if the process always ends (i.e., if there is a maximum value for m), and if this is the case, to know the way in which the process can be slow (i.e., to estimate the maximum possible value of m). Also, we must have a fast way of deciding at execution time if an approximation suffices to get a correctly rounded result. We will tackle this issue in Section 12.4. If a maximum value of m exists, it will be called the *hardness to round*, and the corresponding values of x will be called *hardest-to-round points* (HR points). More formally,

Definition 4 (*Hardness to round.*) *The hardness to round for function f in interval $[a, b]$ is the smallest integer m such that for all floating-point numbers $x \in [a, b]$, either $f(x)$ is a breakpoint or the infinitely precise significand of $f(x)$ is not within 2^{-m} from a breakpoint.*

In 1882, Lindemann showed that the exponential of an algebraic number[5] (possibly complex) different from zero is not algebraic [26]. The machine numbers are rational; thus they are algebraic. From this we deduce that the sine, cosine, exponential, or arctangent of a machine number different from zero cannot be a breakpoint, and the logarithm of a machine number different from 1 cannot be a breakpoint. There is a similar property for functions 2^x, 10^x, $\log_2(x)$ and $\log_{10}(x)$, since if a machine number x is not an integer, then 2^x and 10^x do not have a finite binary representation. Function x^y has many breakpoints, but they are known, so that they can be handled separately if needed [302]. There is a similar result for some of the most frequent algebraic functions [254] (see Section 12.7.1 for a definition of these functions).

Therefore, with the most common functions, for any x (we do not consider the trivial cases such as $\exp(0) = 1$ or $\ln(1) = 0$), there exists m such that the TMD cannot occur. Since there is a finite number of machine numbers x, there exists a value of m such that for all x the TMD does not occur. Unfortunately, this reasoning does not allow us to know what is the order of magnitude of this value of m.

In the following, we are going to try to estimate this value of m. Since the cost of evaluating the elementary functions increases with m, we have to make sure that m is not too large. Before that, let us explain how we can quickly check if we have enough information to guarantee a correctly rounded result.

12.4 Ziv's Rounding Test

Assume that the chosen rounding function is round-to-nearest (ties to even or to away), noted RN, and that we wish to implement function f with correct rounding in radix-2, precision-p floating-point arithmetic. We assume that $f(x)$ is not exactly halfway between two consecutive floating-point numbers. As explained above, with the most common functions, this does not happen, except in rare, known in advance, cases. We assume that we have been able to build an approximation to $y = f(x)$

[5] An algebraic number is a root of a nonzero polynomial with integer coefficients.

with relative error significantly less than 2^{-p}, represented by a double-word number (y_h, y_ℓ), where y_h and y_ℓ are precision-p floating-point numbers. This can be done using a technique similar to the one presented in Example 7, in Section 5.3.3. Hence, we assume

$$y_h + y_\ell = y \cdot (1 + \alpha), \qquad \text{with} \quad |\alpha| \leq \epsilon. \tag{12.2}$$

where ϵ is a known relative error bound (assumed significantly less than 2^{-p}, otherwise it would make no sense returning a double word number, and it would be hopeless trying to return a correctly rounded result). We assume that y_ℓ is small enough in front of y_h, so that $y_h = \mathrm{RN}(y_h + y_\ell)$, otherwise, it suffices to use the Fast2Sum algorithm to get into this situation. We wish to find some way of very quickly checking whether $y_h = \mathrm{RN}(y)$ or not. If we cannot make sure that $y_h = \mathrm{RN}(y)$, we will have to compute another, more accurate, approximation to y. In Ziv's Accurate Portable Mathlib (or `libultim`) library of elementary functions [481], the test being performed to determine if we can be certain that $y_h = \mathrm{RN}(y)$ is the following one:

$$\textit{Is } y_h = \mathrm{RN}\left(y_h + \mathrm{RN}(y_\ell \cdot e)\right)?$$

when e is some cleverly chosen "magic constant". In [135], de Dinechin et al. analyze that test, and show the following result

Theorem 21 *Assume that y_h is a floating-point number such that $\frac{1}{4}\,ulp\,(y_h)$ is in the normal range, and that ϵ is less than $1/(2^{p+1}+1)$. Also assume that $y_h = \mathrm{RN}(y_h + y_\ell)$ and $|(y_h + y_\ell) - y| < \epsilon \cdot |y|$, with $\epsilon < 2^{-p-1}$. If*

$$e \geq \frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}$$

then $y_h = \mathrm{RN}\left(y_h + \mathrm{RN}(y_\ell \cdot e)\right)$ implies $y_h = \mathrm{RN}(y)$.

Hence a natural choice for the magic constant e is

$$\mathrm{RU}\left(\frac{1 + 2^{-p}}{1 - \epsilon - 2^{p+1}\epsilon}\right).$$

Choosing a larger value would lead to too many "wrong alarms", i.e., cases for which $y_h \neq \mathrm{RN}\left(y_h + \mathrm{RN}(y_\ell \cdot e)\right)$ and yet $y_h = \mathrm{RN}(y)$, and choosing a smaller value could lead to incorrect results.

12.5 Some Experiments

Let us now go back to our problem of estimating what is the smallest m such that if the approximation to $f(x)$ is with error less than or equal to 2^{-m}, then rounding the approximation is equivalent to rounding the exact result. Schulte and Swartzlander [417, 418] proposed algorithms for producing correctly rounded results for the functions $1/x$, square root, 2^x, and $\log_2 x$ in binary32 arithmetic. To find the correct value of m, they performed an exhaustive search for $p = 16$ and $p = 24$. For $p = 16$, they found $m = 35$ for $\log_2$ and $m = 29$ for 2^x, and for $p = 24$, they found $m = 51$ for $\log_2$ and $m = 48$ for 2^x. One would like to extrapolate those figures and find $m \approx 2p$. To check that assumption, an exhaustive search was conducted, for some functions and domains, assuming binary64 arithmetic, in the Arénaire project of LIP laboratory (Lyon, France), using algorithms originally designed by

Lefèvre [305, 306]. Before giving the obtained results, let us explain why in practice we always find that the required value of m is around $2p$ (or slightly above $2p$).

12.6 A "Probabilistic" Approach to the Problem

What we are going to do in this section is *not rigorous*: we are going to apply probabilistic concepts to a purely deterministic problem. What we want is just to *understand* why we get $m \approx 2p$ in practice. To simplify the presentation, we assume rounding to the nearest only. Generalization to other rounding modes is straightforward. Such a probabilistic study has been done by Dunham [163] and by Gal and Bachelis [199]. Stehlé and Zimmermann [434] also use this kind of probabilistic argument to implement a variant of Gal's Accurate Tables Method (see Chapter 6).

Let f be an elementary function. We assume in the following that when x is a precision-p floating-point number, the bits of $f(x)$ after the pth position can be viewed as if they were random sequences of zeroes and ones, with probability $1/2$ for 0 as well as for 1. This can be seen as an application of the results of Feldstein and Goodmann [185], who estimated the statistical distribution of the trailing digits of the variables of a numerical computation. For many functions, this assumption will not be reasonable for very small arguments because of the simplicity of their Taylor expansions[6] (for instance, if x is small, then $\exp(x)$ is very close to $x+1$). The case of small arguments must be dealt with separately: we will deal with it in Section 12.8.1. We also assume that the bit patterns (after position p) obtained during computations of $f(x_1)$ and $f(x_2)$ for different values of x_1 and x_2 can be considered "independent." We made the same assumption in Chapter 6 to estimate the cost of building "accurate tables." The infinitely precise significand[7] y of $f(x)$ has the form:

$$y = y_0.y_1y_2\cdots y_{p-1}\overbrace{01111111\cdots11}^{kbits}xxxxx\cdots$$

or

$$y = y_0.y_1y_2\cdots y_{p-1}\overbrace{10000000\cdots00}^{kbits}xxxxx\cdots$$

with $k \geq 1$. What we want to estimate is the maximum possible value of k. That value, added with p, will give the value of m that must be chosen. From our probability assumptions, the "probability" of getting $k \geq k_0$ is 2^{1-k_0}. We assume that there are n_e different possible exponents.[8] Therefore there are $N = 2 \times n_e \times 2^{p-1}$ floating-point numbers. The probability of having at least one input number leading to a value of k greater than or equal to k_0 is:

$$\mathcal{P}_{k_0} = 1 - \left[1 - 2^{1-k_0}\right]^N. \tag{12.3}$$

[6] This should not be viewed as a problem since this will allow us to easily return correctly rounded results for small arguments (see Tables 12.4 and 12.5).

[7] See Section 2.1.4 for a definition.

[8] That number n_e is not necessarily the number of possible exponents of the considered floating-point format. It is the number of different exponents of the set of the input values for which we want to estimate what largest value of k will appear. For instance, if f is the cosine function and we only want to know what will be the largest value of k for input values between 0 and $\pi/2$, we will not consider exponents larger than 1.

Now we are looking for the value k_0 such that the probability of getting at least one value of k greater than k_0 (among the N different floating-point results) should be less than 1/2. $\mathcal{P}_{k_0} \leq 1/2$ as soon as $1/2 \leq \left[1 - 2^{1-k_0}\right]^N$, that is, as soon as

$$\ln \frac{1}{2} \leq N \times \ln \left[1 - 2^{1-k_0}\right]. \tag{12.4}$$

Define t as 2^{1-k_0}. If t is small enough, we can approximate $\ln \left[1 - 2^{1-k_0}\right] = \ln(1-t)$ by $-t$. Therefore (12.4) is roughly equivalent to

$$Nt - \ln 2 \leq 0.$$

Therefore $\mathcal{P}_{k_0} \leq 1/2$ as soon as $2^{1-k_0} \leq \ln 2/N$, that is, when:

$$k_0 \geq 1 - \frac{\ln(\ln 2)}{\ln 2} + \log_2 N \simeq 1.529 + \log_2 N;$$

that is,

$$k_0 \geq p + \log_2(n_e) + 1.529. \tag{12.5}$$

Therefore if we only consider a small number of values of n_e (which is the case in practice for many elementary functions[9]), the maximum value of k encountered will be (on average) slightly greater than p. Since the value of m that must be chosen is equal to p plus the maximum value of k, we deduce that in practice, m must be slightly greater than $2p$. The probabilistic assumptions that we used cannot be proved, but we can try to check them: Table 12.1 shows the actual and expected (from our probabilistic assumptions) number of occurrences of the different values of k for $\sin x$ (with $1 \leq x < 2$) and $p = 24$.

Until we get sure bounds[10] on m, our probabilistic assumptions can help to design algorithms that are very likely to always return correctly rounded results. Let us examine two examples.

- Assume that we want to evaluate exponentials of binary64 floating-point numbers, and let us pretend that we do not already know the hardest-to-round points.[11] The largest possible exponent for the input value is 9 (since $e^{2^{10}}$ is too large to be representable). Moreover, if a number x has an absolute value less than 2^{-53}, then one can easily deduce the value that must be returned (1 in round-to-nearest mode, $1 + \text{ulp}(1)$ if $x > 0$ in round toward $+\infty$ mode, etc., see Tables 12.4 and 12.5). Therefore we must consider 63 possible exponents. Thus the number N of floating-point numbers to be considered in (12.3) is $2 \times 63 \times 2^{52}$. If we apply (12.3), we find that

[9] The exponential of a large number is an overflow, whereas the exponential of a very small number, when rounded to the nearest is 1. Thus there is no need to consider many different exponents for the exponential function. Concerning the trigonometric functions, I think that a general-purpose implementation should ideally provide correctly rounded results whenever the function is mathematically defined. And yet, many may argue that the sine, cosine, or tangent of a huge number is meaningless and I must recognize that *in most cases*, it does not make much sense to evaluate a trigonometric function of a number with a large exponent, unless we know for some reason that that number is exact. Maybe one day this problem will be solved by attaching an "exact" bit to the representation of the numbers themselves, as suggested for instance by Gustafson [211].

[10] For the IEEE-754 binary64/double-precision format, we have obtained the bounds for many functions and domains (see Section 12.8.4). Getting *tight* bounds for much higher precisions seems out of reach. And yet, recent results tend to show that getting *loose* (yet still of interest) bounds for "quad"/binary128 precision might be feasible.

[11] In fact, we know them: see Table 12.6.

Table 12.1 *Actual and expected numbers of digit chains of length k of the form* $1000\cdots0$ *or* $0111\cdots1$ *just after the p-th bit of the infinitely precise significand of sines of floating-point numbers of precision* $p = 24$ *between* $1/2$ *and* 1 *[356].*

k	Actual number of occurrences	Expected number of occurrences
1	4193834	4194304
2	2098253	2097152
3	1048232	1048576
4	522560	524288
5	263414	262144
6	131231	131072
7	65498	65536
8	32593	32768
9	16527	16384
10	8194	8192
11	4093	4096
12	2066	2048
13	1063	1024
14	498	512
15	272	256
16	141	128
17	57	64
18	32	32
19	25	16
20	14	8
21	6	4
22	5	2
23	0	1

  - if $m = 113$ (i.e., $k = 60$), then the probability of having incorrectly rounded values is about 0.6;
  - if $m = 120$, then the probability of having incorrectly rounded values is about 0.007.

  Concerning this function, the worst cases are known (see Table 12.6). For double-precision/binary64 input numbers of absolute values larger than 2^{-30}, $m = 114$ suffices. For input numbers of absolute value less than 2^{-30}, our probabilistic assumptions are no longer valid.
- If we want to evaluate sines and cosines of IEEE-754 binary64 floating-point numbers, the largest possible exponent for the input value is 1023, and if a number x has an absolute value less than 2^{-26} one can easily deduce the values that must be returned (see Tables 12.4 and 12.5). Therefore we must consider 1051 possible exponents. Thus N equals $2 \times 1051 \times 2^{52}$. If we apply (12.3), we find that

  - if $m = 120$ (i.e., $k = 67$), then the probability of having incorrectly rounded values is about 0.12;
  - if $m = 128$ (i.e., $k = 75$), then the probability of having incorrectly rounded values is about 0.0005.

Concerning these functions, we still do not know what is the worst case in the full IEEE-754 binary64 range. We only know worst cases for input values of absolute value less than around 1.570796311 for the cosine function, and less than around 3.141357 for the sine function (see Table 12.11).

We must understand that in the preceding examples "the probability of having incorrectly rounded values" does not mean "given x, the probability that $\exp(x)$ or $\sin(x)$ is not correctly rounded." This last probability is much smaller (in the given cases, it is null or of the order of magnitude of $1/N$). What we mean is "the probability that among *all* the N possible values, *at least one* is not correctly rounded."

12.7 Upper Bounds on m

The probabilistic arguments of the preceding section make it possible to have an *intuition* of the accuracy required during the intermediate computation to get a correctly rounded result. They do not constitute a proof. An exhaustive search has been done for the binary32 format ($p = 24$). Such an exhaustive search is partly completed for binary64 ($p = 53$) [308, 310] (see Section 12.8.4), and a few cases have been found for the decimal64 format [312], and the Intel double-extended precision [433]. However, for larger precisions (e.g., binary128), an exhaustive search is far beyond the abilities of the most powerful current computers. To manage such cases, there is some hope, concerning methods that will give *loose* (yet, hopefully, still of interest) bounds in the forthcoming years. In the meanwhile, let us now try to check if there are results from number theory that can help us. Although they are not really the focus of this book, we first give some results concerning the algebraic functions. Then, we deal with the transcendental functions.

12.7.1 Algebraic Functions

A function f is an algebraic function if there exists a nonzero bivariate polynomial P with integer coefficients such that for all x in the domain of f we have $P(x, f(x)) = 0$. Examples of algebraic functions are $x^2 + 4$, $\sqrt{x}$, $x^{5/4}$. Iordache and Matula [248] gave some bounds on m for the division, the square root, and the square root reciprocal functions. Lang and Muller [295] provided bounds for some other functions. Some of their results are summarized in Table 12.2. Brisebarre and Muller give bounds for functions x^r, where r is a rational number [66].

Although the values given in Table 12.2 are not always optimal, they are small enough to be of practical use. Unfortunately, as we are going to see, this is not the case with the transcendental functions.

12.7.2 Transcendental Functions

A transcendental function is a function that is not algebraic. Almost all functions dealt with in this book (sine, cosine, exponentials, and logarithms of bases 2, e, and 10, hyperbolic functions…) are transcendental.

To the knowledge of the author, the best current result is the following [360]. Before presenting it, let us define some notation: if $\alpha = a/b$ is a rational number, with $b > 0$ and $\gcd(a, b) = 1$, we will define $H(\alpha)$ as $\max\{|a|, b\}$.

Table 12.2 *Some bounds [295] on the size of the largest digit chain of the form* $1000\cdots0$ or $0111\cdots1$ *just after the p-th bit of the infinitely precise significand of* $f(x)$ *(or* $f(x, y)$*), for some simple algebraic functions. An upper bound on m is p plus the number given in this table.*

Function	Size of the largest chain $01111\cdots1$	Size of the largest chain $10000\cdots0$
Reciprocal	$= p$ (p odd)	$= p$
	$\leq p$ (p even)	
Division	$= p$	$= p$
Square root	$= p + 2$	$= p$
$1/\sqrt{x}$	$\leq 2p + 2$	$\leq 2p + 2$
$\sqrt{x^2 + y^2}$ with $\frac{1}{2} \leq x, y < 1$	$= p + 2$	$= p + 2$
$\frac{x}{\sqrt{x^2+y^2}}$ with $\frac{1}{2} \leq x, y < 1$	$\leq 3p + 3$	$\leq 3p + 3$

Theorem 22 (Y. Nesterenko and M. Waldschmidt [360], specialized here to the rational numbers) *Let* α *and* α' *be rational numbers. Let* θ *be an arbitrary nonzero real number. Let* A, A', *and* E *be positive real numbers with*[12]

$$E \geq e, \quad A \geq \max\left(H(\alpha), e\right), \quad A' \geq H(\alpha').$$

Then

$$\begin{aligned} &\left|e^{\theta} - \alpha\right| + |\theta - \alpha'| \geq \\ &\exp\Big\{-211 \cdot \Big(\ln A' + \ln\ln A + 2\ln(E \cdot \max\{1, |\theta|\}) + 10\Big) \\ &\cdot\Big(\ln A + 2E|\theta| + 6\ln E\Big) \cdot \Big(3.7 + \ln E\Big) \cdot \Big(\ln E\Big)^{-2}\Big\}. \end{aligned}$$

Let us try to apply this theorem to find an upper bound on m for the computation of the exponentials of numbers of absolute value less than $\ln(2)$ in precision-p binary floating-point arithmetic. We will consider numbers of absolute value larger than 2^{-p-1} only (since if $|x| \leq 2^{-p-1}$, $\mathrm{RN}(\exp(x)) = 1$, see Section 12.8.1). Hence we will apply Theorem 22 with the following parameters:

- $\alpha' = \theta$, and $|\theta| \in \left[2^{-p-1}, \ln(2)\right]$. As a consequence, the possible irreducible rational representations of $|\theta| = |\alpha'|$ with largest numerator and denominator are of the form

$$\frac{\overbrace{1xxx\cdots x}^{p \text{ bits}}}{2^{2p}},$$

  so that $H(\alpha') \leq 2^{2p}$;
- α is a midpoint between $1/2$ and 2, hence it is of the form

$$\frac{\overbrace{1xxx\cdots x}^{p+1 \text{ bits}}}{2^{p}} \quad \text{or} \quad \frac{\overbrace{1xxx\cdots x}^{p+1 \text{ bits}}}{2^{p+1}}$$

  so that $H(\alpha) \leq 2^{p+1}$;
- $E = e$.

[12] Here, $e = 2.718\cdots$ is the base of the natural logarithm.

We therefore obtain

$$\begin{aligned}&\left|e^{\theta}-\alpha\right| \geq \\ &\exp\Big\{-211\cdot\Big(2p\ln(2)+\ln(p+1)+\ln(\ln(2))+12\Big) \\ &\cdot\Big((p+1)\ln(2)+2e\ln(2)+6\Big)\cdot 4.7\Big\},\end{aligned}$$

i.e.,

$$\begin{aligned}&\left|e^{\theta}-\alpha\right| \geq \\ &2^{\left\{-991.7\cdot\left(2p\ln(2)+\ln(p+1)+\ln(\ln(2))+12\right)\cdot\left((p+1)+2e+\frac{6}{\ln(2)}\right)\right\}},\end{aligned}$$

therefore,

$$\begin{aligned}m &\leq \left\lceil -991.7\cdot\left(2p\ln(2)+\ln(p+1)+\ln(\ln(2))+12\right)\right. \\ &\quad \left.\cdot\left((p+1)+2e+\tfrac{6}{\ln(2)}\right)\right\rceil \\ &\leq \left\lceil 1375p^2+992p\ln(p+1)+32287p+14968\ln(p+1)+174124\right\rceil.\end{aligned}$$

Table 12.3 gives the values of m obtained using this result, for various values of p.

Looking at that table, we may believe that in rare cases, to guarantee correct rounding in binary64 arithmetic, we may need to compute exponentials or logarithms with a number of bits whose order of magnitude is a few millions (around 17 millions for exponentials of quad-precision numbers less than $\ln(2)$), which is feasible but quite expensive: as I am writing these lines, on a good laptop, using algorithms based on the arithmetic-geometric mean (AGM, see Chapter 7), computing exponentials of floating-point numbers with 10 million bits of accuracy takes around 20 seconds. However, we must keep in mind that, following our probabilistic study, the existence of such cases is *extremely unlikely*. For instance, in binary64 arithmetic, we are almost certain that it suffices to evaluate the functions with an intermediate accuracy that is around 120 bits. Hence, although of great theoretical interest, Theorem 22 is rather disappointing for our purpose. The only way to solve this problem is to actually compute the hardest-to-round cases, at least for the most common functions and domains, and/or to isolate some domains (typically, near 0, see Section 12.8.1) where correct rounding is straightforward.

Table 12.3 *Upper bounds on m for various values of* p*, obtained from Theorem 22 and assuming input values between* $-\ln(2)$ *and* $\ln(2)$.

p	m
24	1865828
53	6017142
113	17570144

12.8 Solving the TMD in Practice

12.8.1 Special Input Values

For many functions, we can easily deal with the input arguments that are extremely close to 0. For example, consider the exponential of a very small positive number ϵ, on a precision-p binary floating-point format, assuming rounding to nearest. If $0 \leq \epsilon < 2^{-p}$, then (since ϵ is a p-bit number), $\epsilon \leq 2^{-p} - 2^{-2p}$. Hence,

$$e^{\epsilon} \leq 1 + (2^{-p} - 2^{-2p}) + \frac{1}{2}(2^{-p} - 2^{-2p})^2 + \cdots < 1 + 2^{-p},$$

therefore

$$\exp(\epsilon) < 1 + \frac{1}{2}\ \text{ulp}\ (\exp(\epsilon)).$$

Thus, the correctly rounded value of $\exp(\epsilon)$ is 1. Now, if $-2^{-p+1} \leq \epsilon \leq 0$ then

$$\exp(\epsilon) > 1 + \epsilon \geq 1 - 2^{-p-1}.$$

so that the correctly rounded value of $\exp(\epsilon)$ is 1.

A similar reasoning can be done for other functions and rounding modes. Some results are given in Tables 12.4 and 12.5.

12.8.2 The L-Algorithm

In his Ph.D. dissertation [306], Lefèvre gives an algorithm for finding the hardest-to-round points for a regular enough function. Let us call that algorithm the L-Algorithm. It uses linear approximations of the function and massive parallelism. Some improvements have been introduced in [307]. The reader can find a detailed presentation in [356]. The L-Algorithm has been used for finding the hardest-to-round points given in Tables 12.6, 12.7, 12.8, 12.9, 12.10, 12.11, and 12.12. A fast implementation of the L-Algorithm on GPUs is presented in [193].

This technique was also used by Harrison for building fast and accurate programs for evaluating Bessel functions [223] (guaranteeing good relative accuracy required to locate the floating-point numbers that are nearest a zero of the implemented function).

12.8.3 The SLZ Algorithm

The SLZ algorithm, introduced by Stehlé, Lefèvre, and Zimmermann [431–433], consists of

- approximating the function being considered by degree-d polynomials, in many subintervals, with an accuracy such that finding hardest-to-round points for function f is equivalent to finding hardest-to-round points for the polynomials;
- using a technique due to Coppersmith for finding the hardest-to-round points of each polynomial. Coppersmith's algorithm, mainly used in cryptanalysis, allows one to find small roots of polynomials modulo an integer [110, 111].

Table 12.4 *Some results for small values in the double-precision/binary64 format, assuming* ***rounding to nearest*** *(some of these results are extracted from [356]). These results make finding hardest-to-round points useless for numbers of tiny absolute value. The number* $\alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$ *is approximately equal to* $1.4422\cdots \times 2^{-26}$*, and* $\eta \approx 1.1447 \times 2^{-26}$*. If* x *is a real number, we let* x^- *denote the largest floating-point number strictly less than* x*.*

This function	Can be replaced by	When
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-53}$
$\exp(\epsilon), \epsilon \leq 0$	1	$\lvert\epsilon\rvert \leq 2^{-54}$
$\exp(\epsilon) - 1$	ϵ	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$\exp(\epsilon) - 1, \epsilon \geq 0$	ϵ^+	$\mathrm{RN}(\sqrt{2}) \times 2^{-53} \leq \epsilon < \mathrm{RN}(\sqrt{3}) \times 2^{-52}$
$\mathrm{log1p}(\epsilon) = \ln(1+\epsilon)$	ϵ	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-53}$
$2^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.4426\cdots \times 2^{-53}$
$2^\epsilon, \epsilon \leq 0$	1	$\lvert\epsilon\rvert < 1.4426\cdots \times 2^{-54}$
$10^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.7368 \times 2^{-55}$
$10^\epsilon, \epsilon \leq 0$	1	$\lvert\epsilon\rvert < 1.7368 \times 2^{-56}$
$\sin(\epsilon)$, $\sinh(\epsilon)$, $\sinh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq \alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$
$\arcsin(\epsilon)$	ϵ	$\lvert\epsilon\rvert < \alpha = \mathrm{RN}(3^{1/3}) \times 2^{-26}$
$\sin(\epsilon)$, $\sinh^{-1}(\epsilon)$	$\epsilon^- = \epsilon - 2^{-78}$	$\alpha < \epsilon \leq 2^{-25}$
$\sinh(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\alpha < \epsilon < 2^{-25}$
$\arcsin(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\alpha \leq \epsilon < 2^{-25}$
$\cos(\epsilon)$	1	$\lvert\epsilon\rvert < \gamma = \mathrm{RN}(\sqrt{2}) \times 2^{-27}$
$\cos(\epsilon)$	$1^- = 1 - 2^{-53}$	$\gamma \leq \lvert\epsilon\rvert \leq 1.2247 \times 2^{-26}$
$\cosh(\epsilon)$	1	$\lvert\epsilon\rvert < 2^{-26}$
$\cosh(\epsilon)$	$1^+ = 1 + 2^{-52}$	$2^{-26} \leq \lvert\epsilon\rvert \leq \mathrm{RN}(\sqrt{3}) \times 2^{-26}$
$\cosh(\epsilon)$	$1^{++} = 1 + 2^{-51}$	$\mathrm{RN}(\sqrt{3}) \times 2^{-26} < \lvert\epsilon\rvert \leq 1.118 \times 2^{-25}$
$\tan(\epsilon)$, $\tanh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert \eta = \mathrm{RN}(12^{1/3}) \times 2^{-27}$
$\tanh(\epsilon)$, $\arctan(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq \eta$
$\tan(\epsilon)$, $\tanh^{-1}(\epsilon)$	$\epsilon^+ = \epsilon + 2^{-78}$	$\eta \leq \epsilon \leq 1.650 \times 2^{-26}$
$\arctan(\epsilon)$, $\tanh(\epsilon)$	$\epsilon^- = \epsilon - 2^{-78}$	$\eta < \epsilon \leq 1.650 \times 2^{-26}$

A description of SLZ can be found in [356]. An implementation of SLZ can be found at http://www.loria.fr/~zimmerma/software/.

12.8.4 Nontrivial Hardest-to-Round Points Found So Far

A special case: power functions

Vincent Lefèvre ran the L-Algorithm to find hardest-to-round points in binary64 precision for functions x^n (where n is an integer), for $3 \leq n \leq 733$. Among all these values of n, the hardest-to-round case was attained for $n = 458$. It corresponds to

$$x = 1.00001111001110001100111110101010110010110111100011010$$

for which we have

Table 12.5 *Some results for small values in the double-precision/binary64 format, assuming* ***rounding toward*** $-\infty$ *(some of these results are extracted from [356]). These results make finding hardest-to-round points useless for numbers of tiny absolute value. If x is a real number, we let x^- denote the largest floating-point number strictly less than x. The number $\tau = \mathrm{RN}(6^{1/3}) \times 2^{-26}$ is approximately equal to $1.817\cdots \times 2^{-26}$.*

This function	Can be replaced by	When
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-52}$
$\exp(\epsilon), \epsilon < 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert \leq 2^{-53}$
$\exp(\epsilon) - 1$	ϵ	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-52}$
$\ln(1+\epsilon), \epsilon \neq 0$	ϵ^-	$-2^{-52} < \epsilon \leq \mathrm{RN}(\sqrt{2}) \times 2^{-52}$
$2^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.4426\cdots \times 2^{-52}$
$2^\epsilon, \epsilon < 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert < 1.4426\cdots \times 2^{-53}$
$10^\epsilon, \epsilon \geq 0$	1	$\epsilon < 1.7368 \times 2^{-54}$
$10^\epsilon, \epsilon < 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert < 1.7368 \times 2^{-55}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon > 0$	ϵ^-	$\epsilon \leq \tau = \mathrm{RN}(6^{1/3}) \times 2^{-26}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon \leq 0$	ϵ	$\lvert\epsilon\rvert \leq \tau$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon > 0$	ϵ^{--}	$\tau < \epsilon \leq 2^{-25}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon < 0$	ϵ^+	$\tau < \lvert\epsilon\rvert \leq 2^{-25}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon \geq 0$	ϵ	$\epsilon < \tau$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon < 0$	ϵ^-	$\lvert\epsilon\rvert < \tau$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon \geq 0$	$\epsilon^+ = \epsilon + 2^{-78}$	$\tau \leq \epsilon < 2^{-25}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon < 0$	$\epsilon^{--} = \epsilon - 2^{-77}$	$\tau \leq \lvert\epsilon\rvert < 2^{-25}$
$\cos(\epsilon), \epsilon \neq 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert < 2^{-26}$
$\cosh(\epsilon)$	1	$\lvert\epsilon\rvert < \mathrm{RN}(\sqrt{2}) \times 2^{-26}$
$\cosh(\epsilon)$	$1^+ = 1 + 2^{-52}$	$\mathrm{RN}(\sqrt{2}) \times 2^{-26} \leq \lvert\epsilon\rvert < 2^{-25}$
$\tan(\epsilon), \tanh^{-1}(\epsilon), \epsilon \geq 0$	ϵ	$\epsilon \leq 1.4422\cdots \times 2^{-26}$
$\tan(\epsilon), \tanh^{-1}(\epsilon), \epsilon < 0$	ϵ^-	$\lvert\epsilon\rvert \leq 1.4422\cdots \times 2^{-26}$
$\tanh(\epsilon), \arctan(\epsilon), \epsilon > 0$	ϵ^-	$\epsilon \leq 1.4422\cdots \times 2^{-26}$
$\tanh(\epsilon), \arctan(\epsilon), \epsilon \leq 0$	ϵ	$\lvert\epsilon\rvert \leq 1.4422\cdots \times 2^{-26}$

$$x^{458} = \underbrace{1.00011111000010110000100001110110101110100000000100101}_{\text{53 bits}}\ 1$$
$$\underbrace{00000\cdots00000}_{\text{61 zeros}}\,1110\cdots \times 2^{38}$$

Concerning the "general" power function x^y (where y is not constrained to be an integer), we are far from being able to compute hardest-to-round points. The paper by Lauter and Lefèvre [302] gives a reasonably fast test that allows to check if x^y is a breakpoint. This makes it possible to make sure that Ziv's multilevel strategy terminates.

Hardest-to-round points of transcendental functions in binary64 arithmetic

We have run the L-Algorithm to find hardest-to-round points in binary64/double-precision for the most common functions and domains. The results obtained so far, first partly published in [308] and in a more complete version in [356] are given in Tables 12.6, 12.7, 12.8, 12.9, 12.10, 12.11, 12.12 and 12.13. In these tables, each hardest-to-round point has the following format: the argument (binary64 number) and the truncated result (i.e., the result rounded toward zero in the binary64 format), both in C99's hexadecimal format (described below), and then the bits after the significand of the truncated result. The first bit after the significand is the rounding bit. Then, as these are hard-to-round points,

Table 12.6 *Hardest-to-round points for functions e^x, $e^x - 1$, 2^x, and 10^x. The values given here and the results given in Tables 12.4 and 12.5 suffice to round functions e^x, 2^x and 10^x correctly in the full binary64/double-precision range (for function e^x the input values between -2^{-53} and 2^{-52} are so small that the results given in Tables 12.4 and 12.5 can be applied, so they are omitted here) [308]. Radix-β exponentials of numbers less than $\log_\beta(2^{-1074})$ are less than the smallest positive machine number. Radix-β exponentials of numbers larger than $\log_\beta(2^{1024})$ are overflows.*

Function	Domain	Argument	Truncated result	Trailing bits
exp	$[\log(2^{-1074}), -2^{-36}]$	-1.12D31A20FB38BP5	1.5B0BF3244820AP-50	$0\,1^{58}\,0010\cdots$
		-1.A2FEFEFD580DFP-13	1.FFE5D0BB7EABFP-1	$0\,0^{57}\,1100\cdots$
		-1.ED318EFB627EAP-27	1.FFFFFF84B39C4P-1	$1\,1^{59}\,0001\cdots$
		-1.3475AC05CEAD7P-29	1.FFFFFFECB8A54P-1	$0\,0^{57}\,1001\cdots$
	$[2^{-31}, \log(2^{1025}))$	1.9E9CBBFD6080BP-31	1.000000033D397P0	$1\,0^{57}\,1010\cdots$
		1.83D4BCDEBB3F4P2	1.AC50B409C8AEEP8	$0\,0^{57}\,1000\cdots$
	$(\log(2^{-1074}), -2^{-54}]$	-1.0000000000001P-51	1.FFFFFFFFFFFFCP-1	$0\,0^{100}\,1010\cdots$
	$[2^{-53}, \log(2^{1025}))$	1.FFFFFFFFFFFFFP-53	1.0000000000000P0	$1\,1^{104}\,0101\cdots$
$\exp(x) - 1$	$[2^{-35}, \log(2^{1024}))$	1.274BBF1EFB1A2P-10	1.2776572C25129P-10	$1\,0^{58}\,1000\cdots$
	$(-\infty, -2^{-34}]$	-1.19E53FCD490D0P-23	-1.19E53E96DFFA8P-23	$1\,0^{56}\,1110\cdots$
	$[2^{-51}, \log(2^{1024}))$	1.7FFFFFFFFFFFDP-49	1.8000000000005P-49	$1\,1^{96}\,0110\cdots$
	$(-\infty, -2^{-51}]$	-1.8000000000003P-49	-1.7FFFFFFFFFFFAP-49	$0\,0^{96}\,1000\cdots$
2^x	$(-\infty, +\infty)$	1.12B14A318F904P-27	1.00000017CCE02P0	$0\,0^{58}\,1101\cdots$
		1.BFBBDE44EDFC5P-25	1.0000009B2C385P0	$0\,0^{59}\,1011\cdots$
		1.E4596526BF94DP-10	1.0053FC2EC2B53P0	$0\,1^{59}\,0100\cdots$
10^x	$(0, \log10(2^{1024}))$	1.DF760B2CDEED3P-49	1.0000000000022P0	$1\,0^{58}\,1110\cdots$
		1.A83B1CF779890P-26	1.000000F434FAAP0	$0\,1^{60}\,0101\cdots$
		1.7C3DDD23AC8CAP-10	1.00DB40291E4F5P0	$0\,1^{58}\,0010\cdots$
		1.AA6E0810A7C29P-2	1.4DEC173D50B3EP1	$0\,1^{58}\,0001\cdots$
		1.D7D271AB4EEB4P-2	1.71CE472EB84C7P1	$1\,1^{64}\,0111\cdots$
		1.75F49C6AD3BADP0	1.CE41D8FA665F9P4	$1\,1^{64}\,0101\cdots$
	$(\log10(2^{-1074}), 0)$	-1.1416C72A588A6P-1	1.27D838F22D09FP-2	$1\,1^{65}\,0010\cdots$
		-1.F28E0E25574A5P-32	1.FFFFFFF70811EP-1	$0\,0^{59}\,1011\cdots$

Table 12.7 *Hardest-to-round points for functions* $\ln(x)$ *and* $\ln(1+x)$*. The values given here suffice to round functions* $\ln(x)$ *and* $\ln(1+x)$ *correctly in the full binary64/double-precision range.*

Function	Domain	Argument	Truncated result	Trailing bits
ln	$[2^{-1074}, 2^{-1})$	`1.EA71D85CEE020P-509`	`-1.60296A66B42FFP8`	$1\,1^{60}\,0000\cdots$
		`1.9476E304CD7C7P-384`	`-1.09B60CAF47B35P8`	$1\,0^{60}\,1010\cdots$
		`1.26E9C4D327960P-232`	`-1.4156584BCD084P7`	$0\,0^{60}\,1001\cdots$
		`1.613955DC802F8P-35`	`-1.7F02F9BAF6035P4`	$0\,1^{60}\,0011\cdots$
	$[2^{-1}, 2^{1})$	`1.BADED30CBF1C4P-1`	`-1.290EA09E36478P-3`	$1\,1^{54}\,0110\cdots$
	$[2^{1}, 2^{1025})$	`1.C90810D354618P245`	`1.54CD1FEA76639P7`	$1\,1^{63}\,0101\cdots$
		`1.62A88613629B6P678`	`1.D6479EBA7C971P8`	$0\,0^{64}\,1110\cdots$
$\ln(1+x)$	$[2^{-35}, 2^{98}]$	`1.AB50B409C8AEEP8`	`1.83D4BCDEBB3F3P2`	$1\,1^{60}\,0101\cdots$
		`1.8AA92BC84FF91P54`	`1.2EE70220FB1C4P5`	$1\,1^{60}\,0011\cdots$
		`1.0410C95B580B9P71`	`1.89D56A0C38E6FP5`	$0\,0^{62}\,1011\cdots$
	$[2^{-35}, 2^{1024})$	`1.C90810D354618P245`	`1.54CD1FEA76639P7`	$1\,1^{63}\,0101\cdots$
		`1.62A88613629B6P678`	`1.D6479EBA7C971P8`	$0\,0^{64}\,1110\cdots$
	$(-1, -2^{-35}]$	`-1.7FFFF3FCFFD03P-30`	`-1.7FFFF4017FCFEP-30`	$1\,0^{58}\,1001\cdots$
	$(2^{-51}, 2^{1024})$	`1.8000000000003P-50`	`1.7FFFFFFFFFFFEP-50`	$1\,0^{99}\,1000\cdots$
	$(-1, -2^{-51}]$	`-1.7FFFFFFFFFFFDP-50`	`-1.8000000000001P-50`	$0\,1^{99}\,0110\cdots$

Table 12.8 *Hardest-to-round points for functions* $\log_2(x)$ *and* $\log_{10}(x)$*. The values given here suffice to round functions* $\log_2(x)$ *and* $\log_{10}(x)$ *correctly in the full binary64/double-precision range.*

Function	Domain	Argument	Truncated result	Trailing bits
$\log_2$	$[2^{-1}, 2^{1024})$	`1.B4EBE40C95A01P0`	`1.8ADEAC981E00DP-1`	$1\,0^{53}\,1011\cdots$
		`1.1BA39FF28E3EAP2`	`1.12EECF76D63CDP1`	$0\,0^{53}\,1001\cdots$
		`1.1BA39FF28E3EAP4`	`1.097767BB6B1E6P2`	$1\,0^{54}\,1001\cdots$
		`1.61555F75885B4P128`	`1.00EE05A07A6E7P7`	$1\,1^{53}\,0011\cdots$
		`1.D30A43773DD1BP256`	`1.00DE0E189B724P8`	$1\,0^{53}\,1100\cdots$
		`1.61555F75885B4P256`	`1.007702D03D373P8`	$1\,1^{54}\,0011\cdots$
		`1.61555F75885B4P512`	`1.003B81681E9B9P9`	$1\,1^{55}\,0011\cdots$
$\log_{10}$	$[2^{-1074}, 2^{-1})$	`1.365116686B078P-765`	`-1.CC68A4AEE240DP7`	$0\,1^{61}\,0110\cdots$
		`1.83E55C0285C96P-762`	`-1.CA68A4AEE240DP7`	$0\,1^{61}\,0110\cdots$
		`1.A8639E89F5E46P-625`	`-1.77D933C1A88E0P7`	$1\,1^{61}\,0101\cdots$
		`1.ED8C87C3BF5CFP-49`	`-1.CEE46399392D6P3`	$0\,1^{62}\,0000\cdots$
		`1.27D838F22D0A0P-2`	`-1.1416C72A588A5P-1`	$1\,1^{65}\,0101\cdots$
	$[2^{-1}, 2^{1})$	`1.B0CF736F1AE1DP-1`	`-1.2AE5057CD8C44P-4`	$0\,1^{54}\,0110\cdots$
		`1.89825F74AA6B7P0`	`1.7E646F3FAB0D0P-3`	$1\,0^{57}\,1001\cdots$
	$[2^{1}, 2^{1024})$	`1.71CE472EB84C8P1`	`1.D7D271AB4EEB4P-2`	$0\,0^{64}\,1010\cdots$
		`1.CE41D8FA665FAP4`	`1.75F49C6AD3BADP0`	$0\,0^{66}\,1010\cdots$
		`1.E12D66744FF81P429`	`1.02D4F53729E44P7`	$1\,0^{68}\,1001\cdots$

Table 12.9 *Hardest-to-round points for functions* $\sinh(x)$ *and* $\cosh(x)$*. The values given here suffice to round these functions correctly in the full binary64/double-precision range. If* x *is small enough, the results given in Tables 12.4 and 12.5 can be applied. If* x *is large enough, we can use the results obtained for the exponential function.*

Function	Domain	Argument	Truncated result	Trailing bits
sinh	$\left[2^{-25}, \operatorname{asinh}(2^{1024})\right)$	1.DFFFFFFFFFE3EP-20	1.E000000000FD1P-20	$1\,1^{72}\,0001\cdots$
		1.DFFFFFFFFF8F8P-19	1.E000000003F47P-19	$1\,1^{66}\,0001\cdots$
		1.DFFFFFFFFE3E0P-18	1.E00000000FD1FP-18	$1\,1^{60}\,0001\cdots$
		1.67FFFFFFFD08AP-17	1.680000001AB25P-17	$1\,1^{57}\,0000\cdots$
		1.897374D74DE2AP-13	1.897374FE073E1P-13	$1\,0^{56}\,1011\cdots$
cosh	$\left[2^{-25}, 2^{6}\right)$	1.465655F122FF5P-24	1.000000000000CP0	$1\,1^{61}\,0001\cdots$
		1.7FFFFFFFFFFF7P-23	1.0000000000047P0	$1\,1^{89}\,0010\cdots$
		1.7FFFFFFFFFFDCP-22	1.000000000011FP0	$1\,1^{83}\,0010\cdots$
		1.7FFFFFFFFFF70P-21	1.000000000047FP0	$1\,1^{77}\,0010\cdots$
		1.7FFFFFFFFFDC0P-20	1.00000000011FFP0	$1\,1^{71}\,0010\cdots$
		1.1FFFFFFFFFF0DP-20	1.0000000000A1FP0	$1\,1^{73}\,0110\cdots$
		1.DFFFFFFFFFB9BP-20	1.0000000001C1FP0	$1\,1^{69}\,0010\cdots$
		1.1FFFFFFFFC34P-19	1.000000000287FP0	$1\,1^{67}\,0110\cdots$
		1.7FFFFFFFFF700P-19	1.00000000047FFP0	$1\,1^{65}\,0010\cdots$
		1.DFFFFFFFFEE6CP-19	1.000000000707FP0	$1\,1^{63}\,0010\cdots$
		1.1FFFFFFFFF0D0P-18	1.000000000A1FFP0	$1\,1^{61}\,0110\cdots$
		1.4FFFFFFFFE7E2P-18	1.000000000DC7FP0	$1\,1^{60}\,0011\cdots$
		1.7FFFFFFFFDC00P-18	1.0000000011FFFP0	$1\,1^{59}\,0010\cdots$
		1.AFFFFFFFFCCBEP-18	1.0000000016C7FP0	$1\,1^{58}\,0010\cdots$
		1.DFFFFFFFFB9B0P-18	1.000000001C1FFP0	$1\,1^{57}\,0010\cdots$
		1.EA5F2F2E4B0C5P1	1.710DB0CD0FED5P4	$1\,0^{57}\,1110\cdots$

Table 12.10 *Hardest-to-round points for inverse hyperbolic functions in binary64/double precision. Concerning function* $\sinh^{-1}$*, if the input values are small enough, there is no need to compute the Hardest-to-round points: the results given in Tables 12.4 and 12.5 can be applied.*

Function	Domain	Argument	Truncated result	Trailing bits
asinh	$[2^{-25}, 2^{1024})$	`1.E000000000FD2P-20`	`1.DFFFFFFFFFE3EP-20`	$0\,0^{72}\,1110\cdots$
		`1.E000000003F48P-19`	`1.DFFFFFFFFF8F8P-19`	$0\,0^{66}\,1110\cdots$
		`1.C90810D354618P244`	`1.54CD1FEA76639P7`	$1\,1^{63}\,0101\cdots$
		`1.8670DE0B68CADP655`	`1.C7206C1B753E4P8`	$0\,0^{62}\,1111\cdots$
		`1.62A88613629B6P677`	`1.D6479EBA7C971P8`	$0\,0^{64}\,1110\cdots$
acosh	$[1, 2^{91}]$	`1.297DE35D02E90P13`	`1.3B562D2651A5DP3`	$0\,1^{61}\,0001\cdots$
		`1.91EC4412C344FP85`	`1.E07E71BFCF06EP5`	$1\,1^{61}\,0101\cdots$
	$[1, 2^{1024})$	`1.C90810D354618P244`	`1.54CD1FEA76639P7`	$1\,1^{63}\,0101\cdots$
		`1.62A88613629B6P677`	`1.D6479EBA7C971P8`	$0\,0^{64}\,1110\cdots$

Table 12.11 *Hardest-to-round points for the trigonometric functions in binary64/double precision. So far, we only have hardest-to-round points in the following domains:* $\left[2^{-25}, u\right)$ *where* $u = 1.1001001000011_2 \times 2^1$ *for the sine function* ($u = 3.141357421875_{10}$ *is slightly less than* π); $\left[0, \arccos(2^{-26})\right) \cup \left[\arccos(-2^{-27}), 2^2\right)$ *for the cosine function; and* $\left[2^{-25}, \pi/2\right]$ *for the tangent function. Sines of numbers of absolute value less than* 2^{-25} *are easily handled using the results given in Tables 12.4 and 12.5.*

Function	Domain	Argument	Truncated result	Trailing bits
sin	$\left[2^{-25}, u\right)$	1.E0000000001C2P-20	1.DFFFFFFFFF02EP-20	$0\,0^{72}\,1110\cdots$
		1.E000000000708P-19	1.DFFFFFFFFC0B8P-19	$0\,0^{66}\,1110\cdots$
		1.E000000001C20P-18	1.DFFFFFFFF02E0P-18	$0\,0^{60}\,1110\cdots$
		1.598BAE9E632F6P-7	1.598A0AEA48996P-7	$0\,1^{59}\,0000\cdots$
		1.FE767739D0F6DP-2	1.E9950730C4695P-2	$1\,1^{65}\,0000\cdots$
cos	$\left[2^{-17}, \operatorname{acos}(2^{-26})\right)$ $\cup \left[\operatorname{acos}(-2^{-27}), 2^2\right)$	1.06B505550E6B2P-9	1.FFFFBC9A3FBFEP-1	$0\,0^{58}\,1100\cdots$
		1.34EC2F9FC9C00P1	-1.7E2A5C30E1D6DP-1	$0\,1^{58}\,0110\cdots$
	$\left[0, \operatorname{acos}(2^{-26})\right)$ $\cup \left[\operatorname{acos}(-2^{-27}), 2^2\right)$	1.8000000000009P-23	1.FFFFFFFFFFF70P-1	$0\,0^{88}\,1101\cdots$
tan	$\left[2^{-25}, \pi/2\right)$	1.DFFFFFFFFFF1FP-22	1.E000000000151P-22	$0\,1^{78}\,0100\cdots$
		1.DFFFFFFFFFC7CP-21	1.E000000000545P-21	$1\,1^{72}\,0100\cdots$
		1.DFFFFFFFFF1F0P-20	1.E000000001517P-20	$1\,1^{66}\,0100\cdots$
		1.67FFFFFFFE845P-19	1.6800000002398P-19	$0\,1^{63}\,0100\cdots$
		1.DFFFFFFFFC7C0P-19	1.E00000000545FP-19	$1\,1^{60}\,0100\cdots$
		1.67FFFFFFFA114P-18	1.6800000008E61P-18	$1\,1^{57}\,0100\cdots$
		1.50486B2F87014P-5	1.5078CEBFF9C72P-5	$1\,0^{57}\,1001\cdots$

Table 12.12 *Hardest-to-round points for the inverse trigonometric functions in binary64/double precision. Concerning the arcsine function, the results given in Tables 12.4 and 12.5 and in this table make it possible to correctly round the function in its whole domain of definition.*

Function	Domain	Argument	Truncated result	Trailing bits
asin	$[2^{-25}, 1]$	1.DFFFFFFFFF02EP-20	1.E0000000001C1P-20	$1\,1^{72}\,0001\cdots$
		1.DFFFFFFFFC0B8P-19	1.E000000000707P-19	$1\,1^{66}\,0001\cdots$
		1.DFFFFFFFF02E0P-18	1.E0000000001C1FP-18	$1\,1^{60}\,0001\cdots$
		1.67FFFFFFE54DAP-17	1.6800000002F75P-17	$1\,1^{57}\,0000\cdots$
		1.C373FF4AAD79BP-14	1.C373FF594D65AP-14	$1\,0^{57}\,1010\cdots$
		1.E9950730C4696P-2	1.FE767739D0F6DP-2	$0\,0^{64}\,1000\cdots$
acos	$[2^{-26}, 1]$	1.7283D529A146EP-19	1.921F86F3C82C5P0	$0\,0^{58}\,1000\cdots$
		1.FD737BE914578P-11	1.91E006D41D8D8P0	$1\,1^{62}\,0010\cdots$
		1.1CDCD1EA2AD3BP-9	1.919146D3F492EP0	$1\,1^{57}\,0010\cdots$
		1.60CB9769D9218P-8	1.90BEE93D2D09CP0	$0\,0^{57}\,1011\cdots$
		1.53EA6C7255E88P-4	1.7CDACB6BBE707P0	$0\,1^{57}\,0101\cdots$
	$[-1, -2^{-27}]$	-1.52F06359672CDP-2	1.E87CCC94BA418P0	$1\,0^{56}\,1101\cdots$
		-1.124411A0EC32EP-5	1.9AB23ECDD436AP0	$1\,0^{56}\,1101\cdots$
atan	$(2^{-25}, +\infty)$	1.E000000000546P-21	1.DFFFFFFFFFC7CP-21	$0\,0^{72}\,1011\cdots$
		1.E000000001518P-20	1.DFFFFFFFFF1F0P-20	$0\,0^{66}\,1011\cdots$
		1.E000000005460P-19	1.DFFFFFFFFC7C0P-19	$0\,0^{60}\,1011\cdots$
		1.6800000008E62P-18	1.67FFFFFFFA114P-18	$0\,0^{57}\,1011\cdots$
		1.22E8D75E2BC7FP-11	1.22E8D5694AD2BP-11	$1\,0^{59}\,1101\cdots$
		1.6298B5896ED3CP1	1.3970E827504C6P0	$1\,0^{63}\,1101\cdots$
		1.C721FD48F4418P19	1.921FA3447AF55P0	$1\,0^{58}\,1011\cdots$
		1.EB19A7B5C3292P29	1.921FB540173D6P0	$1\,1^{59}\,0011\cdots$
		1.CCDA26AD0CD1CP47	1.921FB54442D06P0	$0\,1^{57}\,0111\cdots$

Table 12.13 *Hardest-to-round points for functions* $\mathrm{sinpi}(x) = \sin(\pi x)$ *and* $\mathrm{asinpi}(x) = \frac{1}{\pi}\arcsin(x)$ *in binary64/double precision. The domains considered here suffice for implementing these functions in the full binary64 range: for instance if* $x < 2^{-57}$*, a simple continued fraction argument shows that* $\mathrm{RN}(\mathrm{sinpi}\,(x) = \mathrm{RN}(\pi x)$*. The computation of* $\mathrm{RN}(\pi x)$ *is easily performed (see [67]).*

Function	Domain	Argument	Truncated result	Trailing bits
sinpi	$[2^{-57}, 1]$	`1.3C059D39F1D61P-44`	`1.F067F55743EA3P-43`	$1\,1^{56}\,0000\cdots$
		`1.F339AB57731D3P-50`	`1.88173243FB0F4P-48`	$0\,1^{56}\,0010\cdots$
		`1.BC03DF34E902CP-54`	`1.5CBA89AF1F855P-52`	$0\,0^{58}\,1101\cdots$
asinpi	$[\sin(2^{-57}\pi), 1]$	`1.7718543A5606AP-29`	`1.DD95F913D2D22P-31`	$1\,0^{58}\,1011\cdots$
		`1.F067F55743EA4P-43`	`1.3C059D39F1D61P-44`	$0\,0^{56}\,1001\cdots$
		`1.D57C381F54243P-43`	`1.2AE2325F45589P-44`	$0\,1^{56}\,0111\cdots$
		`1.B2D788A56DA58P-46`	`1.14D4375C40C50P-48`	$1\,1^{56}\,0000\cdots$
		`1.5CBA89AF1F855P-52`	`1.BC03DF34E902BP-55`	$1\,1^{57}\,0111\cdots$

the rounding bit is followed by a long run of zeros or ones; the value in exponent (superscript) is the length of this run. Finally, we give the next four bits.

As said above, the argument and the truncated result are written in C99's hexadecimal format for concision. Here, each datum consists of:

- a significand, written as a possible "−" if the number is negative, followed by a "1", followed by a 13-hexadecimal digit number (hence the $1+4\times 13 = 53$ bits of precision), the digits being denoted `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, and `F`;
- the letter `P` (as a separator);
- a binary exponent E, written in decimal, i.e., the significand is to be multiplied by 2^E to get the value.

For instance, the sixth line of Table 12.6 contains the argument

$$1.83D4BCDEBB3F4P2.$$

That number is equal to

$$\left(1+\frac{83D4BCDEBB3F4_{16}}{2^{52}}\right)\cdot 2^2 = \left(1+\frac{2319195600303092_{10}}{2^{52}}\right)\cdot 2^2 = \frac{1705698806918397}{2^{48}},$$

whose exponential is equal to $y\cdot 2^8$, with (in binary)

$$\begin{aligned} y = {} & 1.10101100010100001011010000001001111001000101011 \\ & 10111000000000000000000000000000000000000000000000000000 \\ & 00000000000100010110011100011 0011\cdots \end{aligned}$$

If a and b belong to the same "binade" (they have the same sign and satisfy $2^p \le |a|, |b| < 2^{p+1}$, where p is an integer), let us call their *significand distance* the distance

$$\frac{|a-b|}{2^p}.$$

For instance, the significand distance between 14 and 15 is $(15-14)/8 = 1/8$. Tables 12.6, 12.7, 12.8, 12.9, 12.10, 12.11 and 12.12 allow one to deduce properties such as the following ones [308].

Theorem 23 (Computation of exponentials) *Let y be the exponential of a binary64 number x. Let y^* be an approximation to y such that the significand distance*[13] *between y and y^* is bounded by ϵ.*

- *for $|x| \ge 2^{-30}$, if $\epsilon \le 2^{-53-59-1} = 2^{-113}$, then for any of the four rounding modes, rounding y^* is equivalent to rounding y;*
- *for $2^{-54} \le |x| < 2^{-30}$, if $\epsilon \le 2^{-53-104-1} = 2^{-158}$ then rounding y^* is equivalent to rounding y;*

The case $|x| < 2^{-54}$ is easily dealt with using Tables 12.4 and 12.5.

[13] If one prefers to think in terms of relative error, one can use the following well-known properties: in radix-2 floating-point arithmetic, if the significand distance between y and y^* is less than ϵ, then their relative distance $|y-y^*|/|y|$ is less than ϵ. If the relative distance between y and y^* is less than ϵ_r, then their significand distance is less than $2\epsilon_r$.

Theorem 24 (Computation of logarithms). *Let y be the natural (radix-e) logarithm of a binary64 number x. Let y^* be an approximation to y such that the significand distance between y and y^* is bounded by ϵ. If $\epsilon \leq 2^{-53-64-1} = 2^{-118}$ then for any of the 4 rounding modes, rounding y^* is equivalent to rounding y.*

Conclusion

We have shown that correct rounding of the elementary functions in binary32 or binary64 floating-point arithmetic (at least in some domains for this last case) is possible. It may seem expensive if we think in terms of worst case delay,[14] but we have to consider that using Ziv's multilevel strategy, evaluating an elementary function with correct rounding usually requires the time needed to evaluate it with an intermediate precision of slightly more than p bits, that is, the time already needed by current libraries to compute an elementary function without guaranteed correct rounding.

SUN's Libmcr former library[15] and LIP-Arenaire's CRLIBM[16] libraries provide correct rounding and have reasonably good performances. The Metalibm project[17] [70] provides a tool for the automatic or partly automatic implementation of mathematical functions, that can be used for generating correctly rounded functions. GNU MPFR (see http://www.mpfr.org) is a C multiple precision library that provides correctly rounded functions.

Due to the knowledge of the hardest-to-round points, computing correctly rounded elementary functions will become easier during the next few years. It is therefore high time to think about fully specifying these functions in a floating-point standard (the 2008 version of IEEE 754 only recommends correct rounding of some functions in an appendix). Among the many questions that could be raised when elaborating a standard, there are:

- should the standard provide correct rounding only, or should it also provide "cheaper" rounding modes for applications where speed prevails?
- what can be done for formats wider than binary64, for which it is unlikely that hardest-to-round points can be obtained in the forthcoming years?
- if "cheaper" rounding modes are provided, should a flag be raised when the result is not correctly rounded?
- providing correctly rounded trigonometric functions in the entire floating-point range might be expensive: the domain where the functions are correctly rounded should be discussed.

These issues have been discussed by de Dinechin and Gast [132]. Defour, Hanrot, Lefèvre, Muller, Revol and Zimmermann [146] also presented some aspects of what a standard for the implementation of the mathematical functions could be.

[14] But not that much: de Dinechin et al. achieve a worst case overhead within a factor 2 to 10 of the best libms [131].

[15] Still findable on the Internet.

[16] See https://lipforge.ens-lyon.fr/projects/crlibm/.

[17] See http://lipforge.ens-lyon.fr/www/metalibm/.

Miscellaneous 13

13.1 Exceptions

The handling of the exceptional cases (underflow, overflow, Not A Number, "inexact" flag…) requires even more caution with the elementary functions than with the basic operations $+$, $-$, $\times$, $\div$, and the square root. This is due to the high *nonlinearity* of the elementary functions: when one obtains $+\infty$ as the result[1] of a calculation that only contains the four basic operations and the square root, this does not necessarily mean that the exact result is infinite or too large to be representable, but at least the exact result is likely to be fairly large. Similarly, when one obtains 0, the exact result is likely to be small.[2] With the elementary functions, this is not always true. Consider the following examples.

- Although when X is $+\infty$, the only reasonable value to return when computing $\ln(X)$ is $+\infty$, this may lead to computed results far from the actual results. For instance, in the IEEE-754 binary64 format, the computation of ln(exp(750)) will give $+\infty$, whereas the correct answer should have been 750;
- similarly, when X is equal to 0, the only reasonable value to be returned as $\ln(X)$ is $-\infty$. This may also lead to computed values far from the exact results.

Furthermore, whereas the four arithmetic operations are almost always mathematically defined (the only exception is division by zero, and, incidentally, it is *actually* defined in the IEEE-754 standard), there are many more values for which the elementary functions are not defined or may underflow or

[1] Here, we mean $+\infty$ as the result of an overflow, not the "exact" $+\infty$ that results for instance from a division by zero.

[2] And yet, this is not always true. Consider the following example, due to Lynch and Swartzlander [325] Let us compute

$$f(x) = \frac{x^2}{\sqrt{x^3+1}}.$$

If x is large enough but not too large, the computation of x^3 returns $+\infty$, whereas x^2 is still finite. As a consequence, the returned result is 0, and the exact result is large (close to $\sqrt{x}$).

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_13

overflow, and handling these cases is not always simple. Let us consider the example of the tangent function. The value $\tan(x)$ is infinite if x is $\pi/2$ plus an integer multiple of π. In practice, this never occurs if x is a floating-point number: "machine numbers" are rational numbers, so they cannot be of the form $\pi/2+k\pi$. And yet, a natural question arises: are there machine numbers x so close to a number $\pi/2 + k\pi$ that their tangent is too large to be represented in the floating-point format being used? We can use the results presented in Chapter 11, where we saw (Table 11.2) that the IEEE binary64 number closest to a multiple of $\pi/2$ is $6381956970095103 \times 2^{+797}$, whose tangent is $-2.13\cdots \times 10^{18}$, to see that the problem can never occur in the IEEE binary64 format.[3] The same argument shows that the sine or cosine of a normalized binary64 floating-point number cannot be a subnormal number. However, this does not mean that this problem will never occur in other floating-point formats.

13.1.1 NaNs

Each time $f(x)$ is not mathematically defined and cannot be uniquely defined using continuity, NaN should be returned.[4] Examples are $\sin(\pm\infty)$, $\cos\pm\infty$, or $\ln(-1)$. We must keep in mind that NaN means "Not a Number," which is quite different from "the system is unable to deliver the right result." For instance, it cannot be used to compensate for the lack of a careful range reduction. There are still too many systems that return an error message or a NaN when the sine or cosine of a large number is requested. If the correct result does exist, its adequate representation (normalized or subnormal number, $\pm\infty$, ± 0) must be returned. In [265], W. Kahan gives examples of functions for which there are differences of opinion about whether they should be invalid or defined by convention at internal continuities. One of these examples is 1.0^{∞}, for which Kahan suggests NaN. Another example for which deciding which answer is the right one is not straightforward is function sinpi, defined as $\mathrm{sinpi}(x) = \sin(\pi x)$. The first idea that springs in mind, when considering the definition of that function at argument $\pm\infty$, is to choose $\mathrm{sinpi}(\pm\infty) = \mathrm{NaN}$, and I tend to favor that idea. This is also what is recommended by the appendix to the IEEE 754-2008 standard (see [245], Table 9.1). However, since any large enough floating-point number is an even integer, for all big enough values of $|x|$, $\mathrm{sinpi}(x)$ will be zero. Hence, to limit a possibly problematic change of behavior of a numerical program when x overflows, one may legitimately find it safer to define $\mathrm{sinpi}(\pm\infty) = 0$ (and, similarly, $\mathrm{cospi}(\pm\infty) = 1$). A similar problem occurs for function $(-1)^x$ since, again, for any large enough floating-point number x, x is an even integer so that $(-1)^x = 1$: I tend to favor returning NaN, but many authors (including the writers of the appendix to IEEE 754-2008, see [245] page 44) suggest to return 1.

[3] And it never occurs in the IEEE binary32 or binary128 formats either.

[4] An exception is 0^0. There seems to be a general consensus to return 1 (mainly for reasons of "mathematical beauty": some mathematical relations that are true for all the integers but 0 remain true for 0 with this convention; this is discussed in [227]), although it cannot really be defined using continuity: for any positive number y there exist a sequence (u_n) and a sequence (v_n), both going to zero as n goes to infinity, and such that $u_n^{v_n}$ goes to y. One can choose, for instance, $u_n = 1/n$ and $v_n = -\ln(y)/\ln(n)$. Goldberg [206] justifies the choice $0^0 = 1$ by noticing that if f and g are *analytical* functions that take on the value 0 at 0, then

$$\lim_{x\to 0} f(x)^{g(x)} = 1.$$

13.1.2 Exact Results

It is difficult to know when functions such as x^y give an exact result (and yet, Lefèvre and Lauter [302] show that this problem is less difficult than expected). Using a theorem due to Lindemann,[5] one can show that the sine, cosine, tangent, exponential, or arctangent of a nonzero finite machine number, or the logarithm of a finite machine number different from 1 is not a machine number, so that its computation in floating-point arithmetic is always inexact. For some algebraic functions, the "exact cases" can be listed (see [254]), and for more complicated functions, such as erf, gamma, Bessel, etc., we know (almost) nothing. Let us assume a radix-2, precision-p, floating-point system of extremal exponents $e_{\min}$ and $e_{\max}$, and with subnormal numbers allowed. With the most common functions, the only "exact" operations are:

- for the radix-e logarithm and exponential functions:

  1. $\ln(0) = -\infty$[6];
  2. $\ln(+\infty) = +\infty$;
  3. $\ln(1) = 0$;
  4. $e^0 = 1$;
  5. $e^{-\infty} = 0$;
  6. $e^{+\infty} = +\infty$;

- for the radix-2 logarithm and exponential functions:

  1. $\log_2(0) = -\infty$;
  2. $\log_2(+\infty) = +\infty$;
  3. $\log_2(1) = 0$;
  4. for any integer k such that 2^k is exactly representable (i.e., k is between $e_{\min} - p + 1$ and $e_{\max}$), $\log_2(1.0 \times 2^k) = k$;
  5. $2^0 = 1$;
  6. $2^{-\infty} = 0$;
  7. $2^{+\infty} = +\infty$;
  8. for any integer k between $e_{\min} - p + 1$ and $e_{\max}$, $2^k = 1.0 \times 2^k$;

- for the sine, cosine, tangent, and arctangent functions:

  1. $\sin(0) = 0$;
  2. $\cos(0) = 1$;
  3. $\tan(0) = 0$;
  4. $\arctan(0) = 0$;

- for the hyperbolic sine, cosine, tangent, and arctangent functions:

  1. $\sinh(0) = 0$;
  2. $\sinh(-\infty) = -\infty$;

[5] That theorem, already used in the previous chapter, shows that if x is a nonzero algebraic number, then $\exp(x)$ is transcendental.

[6] Gal and Bachelis [199] suggest returning NaN when $\ln(-0)$ is computed, since -0 can be obtained from a negative underflow (i.e., the exact result can be a nonzero negative number). I prefer to behave as if the input value were exact.

3. $\sinh(+\infty) = +\infty$;
4. $\cosh(0) = 1$;
5. $\cosh(-\infty) = +\infty$;
6. $\cosh(+\infty) = +\infty$;
7. $\tanh(0) = 0$;
8. $\tanh(-\infty) = -1$;
9. $\tanh(+\infty) = 1$;
10. $\tanh^{-1}(0) = 0$;
11. $\tanh^{-1}(-1) = -\infty$;
12. $\tanh^{-1}(1) = +\infty$.

13.2 Notes on x^y

The power function $f(x, y) = x^y$ is very difficult to implement if we want good accuracy [93, 425]. A straightforward use of the formula

$$x^y = \exp(y \ln(x))$$

is to be avoided by all means (unless it is used with a much larger precision than the target precision). First, it would always produce NaNs for negative values of x, although x^y is mathematically defined when $x < 0$ and y is an integer.[7] Second, unless the intermediate calculations are performed with a significantly larger precision, that formula may lead to very large relative errors. Assume that $y \ln(x)$ is computed with relative error ϵ, that is, that the computed value g of $y \ln(x)$ satisfies:

$$g = (1 + \epsilon) y \ln(x).$$

If we neglect the error due to the exponential function itself, we find that the computed value of x^y will be:

$$\begin{aligned} & \exp(g) \\ = \; & \exp(y \ln(x) + y\epsilon \ln(x)) \\ = \; & x^y \times e^{\epsilon y \ln(x)}. \end{aligned}$$

Therefore, the relative error on the result,

$$\left| e^{\epsilon y \ln(x)} - 1 \right| > \epsilon y \ln(x)$$

can be very large.[8] We must realize that even if the exp and ln functions were correctly rounded (which is not yet guaranteed by most existing libraries; see Chapter 12 for a discussion on that problem), the

[7] We should notice that for this function, although it is the best that can be done, returning the machine number that best represents the exact result— according to the active rounding mode — can be misleading: there may be some rare cases where $x < 0$ and y is an inexact result that is, by chance, approximated by an integer. In such cases, returning a NaN would be a better solution, but in practice, we do not know whether y is exact. This problem (and similar ones) could be avoided in the future by attaching an "exact" bit flag in the floating-point representation of numbers, as suggested for instance by Gustafson [211].

[8] Some programming languages avoid this problem by simply not offering function x^y. Of course, this is the best way to make sure that most programmers will use the bad solution $\exp(y \ln(x))$.

Table 13.1 *Error, expressed in ulps, obtained by computing* x^y as $\exp(y\ln(x))$ *for various* x *and* y *assuming that* exp *and* ln *are correctly rounded to the nearest, in IEEE-754 binary64 floating-point arithmetic. The worst case found during our experimentations was* 1200.13 *ulps for* $3482^{3062188649005575/2^{45}}$, *but it is almost certain that there are worse cases.*

x^y	Error in ulps
$(113/64)^{2745497944039423/2^{41}}$	1121.39
2^{994}	718.00
3^{559}	955.17
5^{441}	790.61
56^{156}	1052.15
100^{149}	938.65
220^{124}	889.07
2993^{87}	898.33
$3482^{3062188649005575/2^{45}}$	1200.13

error could be large. Table 13.1 gives the error obtained by computing x^y as $\exp(y\ln(x))$ in IEEE-754 binary64 arithmetic for various values of x and y and assuming that exp and ln are correctly rounded to the nearest.

Now, to estimate the accuracy with which the intermediate calculation must be carried out, we have to see for which values x and y the number $y\ln(x)$ is maximum. It is maximum when x^y is the largest representable number, that is,

$$x^y \approx 2^{e_{\max}+1},$$

which gives $y\ln(x) = (e_{\max}+1)\ln(2)$. If we want a relative error on the result less than α, we must compute $y\ln(x)$ with a relative error less than ϵ, where

$$\left|e^{y\epsilon\ln(x)} - 1\right| < \alpha,$$

which is approximately equivalent to

$$y\epsilon\ln(x) < \alpha.$$

This gives

$$\epsilon < \frac{\alpha}{(e_{\max}+1)\ln(2)}.$$

Therefore we must have

$$-\log_2(\epsilon) > -\log_2(\alpha) + \log_2(e_{\max}+1) + \log_2(\ln(2)).$$

Thus to get a correct result, we must get $\ln(x)$ with a number of additional bits of precision that is slightly more than the number of exponent bits.

All this shows that, although requiring correct rounding for sin, cos, exp, 2^x, ln, $\log_2$, arctan, tan, and the hyperbolic functions would probably be a good step toward high quality arithmetic environments, such a requirement seems difficult to fulfill for the power function if speed is at stake. The former SUN LIBMCR (see Section 14.5) library provided a correctly rounded power function in binary64 arithmetic, by increasing the precision of the intermediate calculations until we can guarantee correct rounding. Unfortunately, since the hardest-to-round cases are not known for this function, this process

can sometimes be rather slow. This function should at least be *faithfully rounded*, that is, the returned value should be one of the two machine numbers closest to the exact result. In any case, it is important, when x^y is exactly representable in the target format, that it is computed exactly—and we must keep in mind that detecting such "exact" cases can be done at reasonable cost [302].

Concerning exceptional cases for the power function, Sayed and Fahmy have investigated what the power function should return when evaluating expressions such as $(+\infty)^{-0}$ or $(-1)^{+\infty}$ [410]. The appendix of the IEEE 754-2008 standard gives similar recommendations, with, however, a few differences. For instance:

- it recommends implementation of three different functions: function `pown` (for which the second argument is necessarily an integer), function `pow` (supposed as general as possible, and defined on $[-\infty, +\infty] \times [-\infty, +\infty]$), and function `powr` (defined as $\exp(y \ln(x))$. For instance, `powr`(x, y) should return a NaN for $x < 0$ (because $\ln(x)$ is undefined), although `pow`$(-3.0, 2.0)$ should return 9. Also notice the important difference: `pow`$(\pm 0, \pm 0)$ is 1 whereas `powr`$(\pm 0, \pm 0)$ is NaN.
- in the case
$$(-0)^{\text{noninteger } <0},$$
Sayed and Fahmy recommend to return a NaN, whereas the IEEE 754-2008 appendix recommends that `pow`$(\pm 0, y)$ should be $+\infty$.

Notice that integer power functions (i.e., functions $(x, n) \to x^n$) are much simpler to implement than the "general" power function since for these functions, the hardest-to-round cases (in binary64 arithmetic) are known for all values of n of practical interest (up to $n = 733$, see [279]). Furthermore, some specific algorithms have been suggested [279], and we also know very tight bounds on the errors of the classical algorithms that compute these functions [209, 403].

13.3 Special Functions, Functions of Complex Numbers

A huge work on the evaluation of special functions (such as the gamma, erf, Jacobi and Bessel functions) has been done by Cody, who initially wrote a package named FUNPACK [96], and later wrote a more portable package, SPECFUN [100]. Macleod designed a package named MISCFUN for the evaluation of several special functions which are not used often enough to have been included in the standard libraries [326]. Cody also worked on the performance evaluation of programs for these functions [99, 104]. Special functions are of particular interest in physics, chemistry, and statistics. They are out of the scope of this book (although several techniques presented in the previous chapter, especially polynomial and rational approximation, are frequently used to implement these functions). The interested reader can consult recent good books on the topic [205, 118]. Marc Mezzarobba's PdD dissertation [344] and recent papers [343, 87, 299] are also of interest: they show that the building of accurate programs that calculate these functions can be at least partly automated. Incidentally, I strongly advise the reader to have a look at the *Dynamic Dictionnary of Mathematical Functions* (DDMF) [34], accessible at http://ddmf.msr-inria.inria.fr/1.9.1/ddmf. The work done by the DDMF team is impressive.

The evaluation of functions of a complex variable can be done using the real functions. The most usual formulas can be found, for instance, in Reference [28], and a discussion on the definition of these functions can be found in Reference [112]. And yet, a naive use of these formulas will frequently lead to inaccurate functions, wrong branch cuts, and under/overflows during the intermediate computations. Kahan's paper on branch cuts [263] brings this problem to the fore and gives elegant solutions. Hull, Fairgrieve, and Tang give reliable and accurate algorithms for the common complex elementary func-

tions [242, 243]. Cody wrote a package named CELEFUNT [101] for testing a complex elementary function library. That package is outdated now but some ideas introduced by Cody in [101] are still of interest.

One can also define square roots and elementary functions of matrices (for instance, an exponential or cosine of a matrix can be defined by the usual Taylor series). This has many numerical applications, for solving some differential equations. There is a large literature on this domain. For instance, Cheng and others deal with the logarithm of a matrix [82], Higham and Smith give an algorithm for computing cosines of matrices [234], and Iserles and Zanna published a paper on the computation of matrix exponentials [249]. The interested reader should consult Higham's book *Functions of matrices: Theory and computation* [233].

Examples of Implementation 14

This chapter is far from being exhaustive. The implementations it describes may not be the best ones, and certainly will at least slightly vary between when this chapter is written and when you read it. Also, the methods used are not always publicized. Some of the implementations presented here are rather old. My purpose, here, is to show, through some examples, how the various techniques described in this book can be used.

14.1 First Example: The Cyrix FastMath Processor

In the Cyrix Fastmath processor [119, 187], five "core" functions were directly implemented: sine, tangent, arctangent, $2^x - 1$, and $\log_2(x+1)$. The other functions were implemented using the core functions. This simplifies the writing and maintenance of the library of functions, but induces some small penalty in terms of accuracy and speed. The approximations used were shown to be monotonic [188]. For example:

- to compute $2^x - 1$ on $[-1, 1]$, a rational approximation of $2^{x/2} - 1$ was first computed, and then the identity
$$2^x - 1 = \left(2^{x/2} - 1\right)\left[\left(2^{x/2} - 1\right) + 2\right]$$
was used;
- to compute $\sin(x)$ on $[-\pi/4, +\pi/4]$, an odd polynomial approximation of the form $xP(x^2)$ was used (we explain how such odd or even approximations can be built in Chapter 4). The cosine was obtained[1] as
$$\cos(x) = \pm\sqrt{1 - \sin^2(x)}.$$
On the Cyrix FastMath processor, the square root was almost as fast as division, so there was little penalty from using it;

[1] Incidentally, this shows that the so frequent idea of computing $\sin^2 x + \cos^2 x$ for several x to quickly check if the trigonometric functions are accurate is *very* naive. In the present case, even if the sine function had been very poor — which was not the case — the computed value of $\sin^2 x + \cos^2 x$ would always have been very close to 1.

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4_14

- to compute $\log_2(x+1)$ on $[1/\sqrt{2}-1, \sqrt{2}-1]$, the number

$$g = \frac{x}{x+2}$$

was computed, so that

$$\log_2(1+x) = \log_2\left(\frac{1+g}{1-g}\right)$$

become an *odd* function of g. Then an odd rational approximation of the form $g \times Q(g^2)$ was used. Using an odd approximation reduces the number of multiplications required to evaluate it;
- to compute $\tan(x)$ on $[-\pi/4, +\pi/4]$, an odd polynomial approximation of the form $xP(x^2)/Q(x^2)$ was used;
- to compute $\arctan(x)$ on $[-\pi/32, +\pi/32]$, a five-segment[2] odd rational approximation of the form $xP(x^2)/Q(x^2)$ was used.

All approximations had the general form $xR(x)$, where $R(x)$ is a rational function or a polynomial. In each case, the graph of R is relatively flat and stays well away from zero, so R can be efficiently and accurately evaluated in fixed-point.

14.2 The INTEL Functions Designed for the Itanium Processor

As explained by Harrison, Kubaska, Story, and Tang [224], the IA-64 instruction set, whose first implementation is the INTEL/HP Itanium processor, has several key features that can be used for designing fast and/or accurate elementary function algorithms:

- some parallelism is available, since there are several floating-point units, and each of them is pipelined;
- the internal INTEL extended-precision format can be used to make binary64 programs very accurate (notice that this is no longer the case if the SSE2 instruction set is used);
- the fused multiply–add instruction (see Section 2.1.5) makes polynomial evaluation faster and in general slightly more accurate than what we would get by using separate additions and multiplications. This and the parallelism also make the latency of the "computational part" of the function evaluation (mainly polynomial evaluation) much shorter than memory references, which must be taken into account: large tables should be avoided whenever it is possible.

The designers of Intel's library for the IA-64 architecture [115, 224, 436] therefore made the following choices:

- to avoid loading constants, a very simple range reduction technique is used;
- polynomials of large degree are favored: this allows the domain where they approximate well enough the function to be large (which simplifies range reduction and requires less memory), and this is not a large penalty in terms of speed (since Estrin's method or variants — see Section 5.2.2 — can be used thanks to the available parallelism), nor in terms of accuracy when binary64 arithmetic is at stake (thanks to the availability of an internal extended-precision format).

[2]The interval was split into five subintervals, and a different approximation was chosen for each subinterval.

The latency of their binary64 functions varies from 52 cycles (for the natural logarithm) to 70 cycles (for the sine or cosine). The accuracy of most of their functions is within 0.51 ulps. The number of double-extended table entries required varies from none at all (tangent and arctangent) to 256 (natural logarithm).

Let us now give two examples, drawn from [224].

14.2.1 Sine and Cosine

$\sin(x)$ is computed as follows

1. range reduction: compute the closest integer N to $16x/\pi$, then compute

$$r = (x - NP_1) - NP_2$$

   using two consecutive fused multiply–adds, where P_1 and P_2 are extended-precision numbers such that $x - NP_1$ is exactly computed and $P_1 + P_2$ is as close as possible to $\pi/16$. This is a variant of Cody and Waite's method — see Section 11.2 —, made more accurate by the availability of the fused multiply–add instruction and the internal extended-precision format;[3]
2. polynomial approximation: we compute two polynomials, an odd polynomial

$$p(r) = r + p_1 r^3 + p_2 r^5 + \cdots + p_4 r^9$$

   that approximates $\sin(r)$ and an even polynomial

$$q(r) = q_1 r^2 + q_2 r^4 + \cdots + q_4 r^8$$

   that approximates $\cos(r) - 1$. Such approximations with particular forms are computed using methods similar to those presented in Chapter 4;
3. reconstruction: the returned result is

$$Cp(r) + (S + Sq(r))$$

   where C is $\cos(N\pi/16)$ and S is $\sin(N\pi/16)$, read from a table.

   The cosine is computed very similarly: it suffices to add 8 to N once it is obtained.

14.2.2 Arctangent

No range reduction is performed. Only two cases are considered: $|x| \geq 1$ and $|x| < 1$. Having such large intervals requires the use of polynomials of large degree. Indeed, a polynomial of degree 47 is used. The case of input arguments greater than 1 is dealt with using

[3] It could probably be made even more accurate, using the technique of Boldo, Li and Daumas (see Section 11.2.2 page 241), which was not known at the time that library of functions was designed.

$$\arctan(x) = \text{sign}\,(x)\frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) \tag{14.1}$$

but since a direct use of (14.1) might sometimes lead to inaccuracies and requires a time-consuming division, some care is taken. More precisely:

1. If $|x| < 1$, the authors of [224] suggest to approximate $\arctan(x)$ by a polynomial of the form

$$x + x^3\left(p_0 + p_1 y + p_2 y^2 + \cdots + p_{22} y^{22}\right)$$

   where $y = x^2$. Again, such approximations with particular forms are computed using methods similar to those presented in Chapter 4.
2. If $|x| \geq 1$, $\arctan(x)$ is approximated by

$$\text{sign}\,(x)\frac{\pi}{2} - c^{45} r(\beta) q(x)$$

   where

   - c is an approximation to $1/x$ (with relative error less than $2^{-8.886}$) obtained using the `frcpa` instruction that is available on the Itanium;
   - $\beta = 1 - xc$ is close to 0;
   - $r(\beta) = 1 + r_1\beta + r_2\beta^2 + \cdots + r^{10}\beta^{10}$ is a degree-10 polynomial approximation to $(1-\beta)^{-45}$;
   - $q(x) = q_0 + q_1 y + q_2 y^2 + \cdots + q_{22} y^{22}$, with $y = x^2$, is a degree-44 polynomial approximation to $x^{45}\arctan(1/x)$.

14.3 The LIBULTIM Library

The LIBULTIM library (also called MathLib) was developed by Ziv and colleagues at IBM. Although it is no longer supported, that library is of great historical importance, since it was the first one that provided correctly rounded transcendental functions. Since at the time it was developed, the hardest-to-round points for correct rounding in binary64 arithmetic were not known (they still are unknown for some functions and domains), the authors assumed that performing the intermediate calculations with 800 bits of precision was enough, which is almost certainly true according to the probabilistic model presented in Section 12.6. However, computing with 800 bits of precision is an overkill: it makes the library of functions very slow at times. Of course, all calculations were not performed with such a huge precision, Ziv's "multilevel strategy" [481] (see Chapter 12) was used: the function being considered was first evaluated with rather low precision, and the precision was increased in case the previous calculation did not suffice to guarantee correct rounding.

14.4 The CRLIBM Library

The CRLIBM library[4] [124, 125] was developed by the Arenaire[5] research group in Lyon, France. It aims at returning correctly rounded results in binary64 arithmetic, for the four following rounding functions: toward $-\infty$, toward $+\infty$, toward 0, and to the nearest ties-to-even. CRLIBM uses the hardest-to-round values that have been computed using Lefèvre's algorithms [306, 308] (see Chapter 12). The first functions of CRLIBM were written by Defour during the preparation of his Ph.D. [144], and most of the current library was written by Lauter during the preparation of his PhD [301, 136]. The writing of the software was coordinated by de Dinechin.The tools Sollya and Gappa have been used for building the polynomial approximations, and computing and certifying the error bounds [85, 86, 134].

Ziv's multilevel strategy is used, but to guarantee correct roundings, two steps only are necessary here: the knowledge of the hardest to round points allows us to avoid overestimating the necessary intermediate precision. More precisely:

- during the first step, the function is evaluated with between 60 and 80 bits of accuracy (depending on the function). The result of this first evaluation is represented as a double-word number (see Section 2.2.2), i.e., as the unevaluated sum of two binary64 numbers. In most cases, this suffices to return a correctly rounded final result. Ziv's rounding test, presented in Section 12.4, is used to check if this is the case. In the following, we call this step the *quick step* of the algorithm;
- when the quick step does not suffice, we use a more accurate yet slower method, tightly targeted at the precision given by Lefèvre's cases. In the following, we call this step the *accurate step* of the algorithm.

The authors of CRLIBM published, with each function, a proof of its behavior. In the earlier versions of CRLIBM, an ad hoc multiple-precision library, called SCSLIB [145], was designed for performing the arithmetic operations needed in the accurate step. Later on, Lauter designed triple-word arithmetic operations [300] based on binary64 arithmetic for this purpose. This had two advantages over the previous approach: it was much faster, and made it easier to use information from the first step in the accurate one.

Although that may seem strange at first glance, the accurate step is simpler than the quick step: first, performance is not an important issue for the accurate step, since that step will rarely be taken, second, the SCSLIB library or the triple-word library provide much precision (indeed, significantly more than what is actually needed), so that we do not need sophisticated evaluation schemes to preserve accuracy, and the error analysis is somehow simpler. Also, all the special cases (zeroes, underflows, overflows, NaNs, etc.) are handled before, during the first step. On the other hand, for the quick step, performance is a primary concern, and tight error bounds must be obtained (using for instance methods such as the one presented in Section 5.3.1).

The CRLIBM team is now working on a follow-up: MetaLIBM (see Section 14.7).

Let us now give two examples of CRLIBM functions.

14.4.1 Computation of $\sin(x)$ or $\cos(x)$ (Quick Step)

Assume we wish to evaluate $\sin(x)$ or $\cos(x)$. The trigonometric range reduction algorithm of CRLIBM computes an integer k and a reduced argument y such that

[4] See http://lipforge.ens-lyon.fr/projects/crlibm/.

[5] Now called AriC: http://www.ens-lyon.fr/LIP/AriC/.

$$x = k \cdot \frac{\pi}{256} + y,$$

where the reduced argument y is computed as a "double-word" $y_h + y\ell$ (i.e., the unevaluated sum of two binary64 numbers, see Section 2.2.2), and belongs to $[-\pi/512, +\pi/512]$. Notice that $\pi/512 < 2^{-7}$. That range reduction uses various methods, depending on the magnitude of the initial argument:

- Cody and Waite's method with two constants (the fastest), for very small arguments;
- Cody and Waite's method with three constants (almost as fast) for small arguments;
- Cody and Waite's method with three constants, using a double-word arithmetic and a 64-bit integer format for representing the integer k, for arguments of moderate magnitude;
- Payne and Hanek's algorithm (see Section 11.4) for the largest arguments.

Then, we read from a table

$$\begin{cases} s_h(k) + s_\ell(k) \approx \sin(k\pi/256) \\ c_h(k) + c_\ell(k) \approx \cos(k\pi/256) \end{cases}$$

and we use

$$\begin{cases} \sin(k\pi/256 + y) = \sin(k\pi/256)\cos(y) + \cos(k\pi/256)\sin(y) \\ \cos(k\pi/256 + y) = \cos(k\pi/256)\cos(y) - \sin(k\pi/256)\sin(y) \end{cases}$$

where $\cos(y)$ and $\sin(y)$ are evaluated by first approximating, using small polynomials, two functions of y, $e(y)$ and $f(y)$, defined by $\cos(y) = 1 + e(y)$ and $\sin(y) = (y_h + y\ell)(1 + f(y))$. The values of $e(y)$ and $f(y)$ are returned as binary64 numbers. This gives 14 extra bits of accuracy, so this first step is very accurate.

14.4.2 Computation of $\ln(x)$

Let us assume that the target rounding function is round to nearest, ties to even. The relative error of the quick step is bounded by $2^{-62.6}$. The accurate step is accurate to more than 119 bits, which suffices to guarantee correct rounding (see Table 12.7). Subnormal input numbers are easily handled using for instance

$$\ln(x) = -52 \cdot \ln(2) + \ln\left(2^{52}x\right).$$

Now, if x is a normal positive floating-point number, range reduction is very simple. From

$$x = m_x \cdot 2^{e_x},$$

where m_x is the significant of x and e_x its exponent, the first idea that springs in mind is to compute $\ln(m_x)$ and then to add $e_x \cdot \ln(2)$ to that result. However, this would result in a very large relative error when $e_x = -1$ and $m_x \approx 2$ (as already noticed by various authors, for instance Wong and Goto: see Section 6.5.1 page 133). To overcome this problem, we define numbers y and E as follows:

$$E = \begin{cases} e_x & \text{if } m_x \le \sqrt{2}, \\ e_x + 1 & \text{if } m_x > \sqrt{2}, \end{cases}$$

and

$$y = \begin{cases} m_x & \text{if } m_x \le \sqrt{2}, \\ m_x/2 & \text{if } m_x > \sqrt{2}. \end{cases}$$

We will evaluate $\ln(x)$ as $E \cdot \ln(2) + \ln(y)$. The interval $(1/\sqrt{2}, \sqrt{2})$ where y lies is split into 128 subintervals. More precisely, from the high-order bits of y considered as an index i, a value r_i close to $1/y$ is read from a table of 128 entries. The number

$$z = y \cdot r_i - 1$$

is then computed *exactly* as a double-word number $z_h + z_\ell$. We have

$$\ln(y) = \ln(1 + z) - \ln(r_i).$$

The 128 possible values $\ln(r_i)$ are tabulated, and since z is small enough (less than 2^{-8}), $\ln(1 + z)$ can easily be approximated by a polynomial. The quick step uses a polynomial of the form

$$P_{\text{quick}}(z) = z - \frac{z^2}{2} + c_3 z^3 + c_4 z^4 + c_5 z^5 + c_6 z^6$$

whose coefficients c_3 to c_6 are binary64 numbers (Chapter 4 explains how such polynomials are computed). The polynomial is evaluated as follows

- the part $c_3 z^3 + c_4 z^4 + c_5 z^5 + c_6 z^6$ is evaluated using z_h only. More precisely, we first compute $z_{h,square} = \text{RN}(z_h^2)$, then

$$p_{35} = \text{RN}(c_3 + \text{RN}(z_{h,square} \cdot c_5)),$$

$$p_{46} = \text{RN}(c_4 + \text{RN}(z_{h,square} \cdot c_6)),$$

  $z_{h,cube} = \text{RN}(z_{h,square} \cdot z_h)$, $z_{h,four} = \text{RN}(z_{h,square} \cdot z_{h,square})$, and finally

$$p_{36} = \text{RN}\left(\text{RN}(z_{h,cube} \cdot p_{35}) + \text{RN}(z_{h,four} \cdot p_{46})\right).$$

- the remaining parts are added as follows: we compute $t_1 = \text{RN}(-\frac{1}{2} z_{h,square} + z_\ell)$, $t_2 = \text{RN}(t_1 + p_{36})$, and we finally obtain the value of the polynomial as a double-word number:

$$(p_h, p_\ell) = \text{Fast2Sum}(z_h, t_2).$$

Adding $E \cdot \ln(2) - \ln(r_i)$ to this result is done in double-word arithmetic. Then Ziv's rounding test (see Section 12.4, page 263) is performed to check if we need to perform the accurate step.

The accurate step uses a polynomial of degree 14, of the form

$$P_{\text{accurate}}(z) = z - \frac{z^2}{2} + c'_3 z^3 + c'_4 z^4 + c'_5 z^5 + \cdots + c'_{14} z^{14}.$$

The coefficients c'_3 to c'_9 are double-word numbers, and the coefficients c'_{10} to c'_{14} are binary64 numbers. The computation of $P_{tail} = c'_5 + c'_6 z_h + c'_7 z_h^2 + \cdots + c'_{14} z_h^9$ is first done using Horner's scheme with the usual binary64 arithmetic operations. Then

$$P_{med} = c'_3 + c'_4 z_h + c'_5 z_h^2 + \cdots + c'_9 z_h^6 + P_{tail} z_h^7$$

is evaluated using Horner's scheme in double-word arithmetic. Finally,

$$(z_h + z_\ell) - \frac{1}{2}(z_h + z_\ell)^2 + (z_h + z_\ell)^3 \cdot P_{med}$$

is calculated in triple-word arithmetic.

14.5 SUN's Former LIBMCR Library

The LIBMCR library[6] was developed by K.C. Ng, N. Toda, and others at SUN Microsystems. The first beta version was published in December 2004. It provided correctly rounded functions in binary64 arithmetic.

As an example, let us show how natural logarithms are evaluated by LIBMCR:

- first, the exceptional cases are processed ($\ln(1) = 0$ and $\ln(+\infty) = +\infty$, $\ln(0) = -\infty$, $\ln(negative) = \text{NaN}$);
- then, an "almost correctly rounded" function is called;
- if the result is too close to the middle of two consecutive floating-point numbers,[7] then a multiple-precision program is called with increasing precision until we can guarantee correct rounding.

Let us summarize the "almost correctly rounded" step. Assume we wish to compute $\ln(x)$. First, x is expressed as $r \times 2^n$. Then, from a table, we read a value y such that y is close to r (more precisely, $|y - r| < 2^{-5} + 2^{-12}$) and $\ln(y)$ is very close to a binary64 number (at a distance less than 2^{-24} ulps). This is Gal's accurate-table method, described in Section 6.3. We therefore have

$$\ln(x) = n \ln(2) + \ln(y) + \ln(r/y).$$

Define $s = (r - y)/(r + y)$. $|s|$ is less than 0.01575, and $\ln(r/y)$ is computed using the Taylor approximation

$$\ln\left(\frac{r}{y}\right) = \ln\left(\frac{1+s}{1-s}\right) = 2s + \frac{2}{3}s^3 + \frac{2}{5}s^5 + \cdots + \frac{2}{13}s^{13}.$$

Much care is taken for evaluating the series with 77-bit accuracy.

14.6 The HP-UX Compiler for the Itanium Processor

As noticed by Markstein [332], it is very frequent that a program invokes both the sine and cosine routines for the same argument (e.g., in rotations, Fourier transforms, etc). These routines share many common calculations: range reduction, computation of the sine, and cosine of the reduced argument. Hence, Markstein suggests using this property for performing these common calculations only once. The HP-UX compiler for Itanium [316] has implemented this methodology. The library does not provide

[6] See http://www.sun.com/download/products.xml?id=41797765.

[7] Assuming rounding to the nearest.

correct rounding, and yet its accuracy is, in general, very good (for instance, the largest observed — i.e., not proven — error for trigonometric functions in binary64 arithmetic is 0.502 ulps [316]).

Markstein's algorithms for elementary functions on the Itanium are presented in his excellent book [331]. For instance, $\exp(x)$ is computed as

$$2^m t_n 2^u,$$

where $m = \lfloor x/\ln(2) \rfloor$, n is constituted by the leading L bits of the fractional part of $x/\ln(2)$, u is the remaining part of that fractional part, and t_n is

$$2^{n \cdot 2^{-L}}$$

read from a table. A typical value of L is 10. The number 2^u is approximated by a polynomial.

14.7 The METALIBM Project

When building an elementary function library, writing just one program for each pair (function, format) is not a fully satisfactory solution:

- first, one would like to obtain good results for the various possible rounding functions. Even if we do not aim at correct rounding, if the user selects—for instance—the round-to-$+\infty$ rounding function, it means that he or she wants to be certain that the computed result will be larger than or equal to the exact result. In general that property will not be satisfied if we just take a program designed for giving good results with round-to-nearest and change the rounding function to round-to-$+\infty$ (in some cases we may even get extremely poor results). Hence, we need at least a set of programs for each rounding function;
- we need to provide *portable* functions, which can run on many different platforms. However, to gain performance, it would be advisable also to have versions of the programs that take into account the specificities of a given platform (depth of the pipelines, availability of an FMA, size of cache memory for table-based methods, possible availability of a floating-point format wider than the target format, which would allow one to avoid or to limit the need for double-word or triple-word arithmetic…): the "best" implementation is highly dependent on the technology. Hence, in addition to a fully "general" set of functions, we may want to have a slightly different, specialized, set of functions for each family of processors of commercial importance;
- although I strongly recommend the availability of correctly rounded functions whenever it is possible, there are applications where accuracy and numerical reproducibility are not an important issue, where we know that, anyway, the values that are input to the functions are not very accurate… in such cases, it may make sense to provide programs that are slightly less accurate but faster. Concerning speed, one may also want to have functions that can be efficiently vectorized, and to be able to choose compromises between optimizing throughput and optimizing latency.

Adding to this the fact that we need elementary functions for all available formats (typically binary32, binary64, and binary128, but decimal formats, if implemented on the target system, need elementary functions too), we conclude that we should ideally offer many variants for each of the functions of a typical mathematical library (libm). This represents a huge programming effort.

Furthermore, libms do not contain all functions of practical interest: as noticed in [70], the web page http://project-mathlibs.web.cern.ch/project-mathlibs/mathTable.html mentions all the functions that are used by CERN engineers in their calculations. Only a fraction of that list can be found in a

standard libm: the other functions need to be implemented by programmers who, frequently, do not have expertise in libm development.

All this clearly shows that several hundreds of function programs are needed. Writing these programs, maintaining them, certifying their behavior when critical applications are at stake, making sure that they are quickly—yet, safely!—adapted each time a new processor architecture appears is a huge task. In the medium and long term, the only solution is to *automate* the generation of function programs, so that they can be generated on-demand, and adapted to a specific context. Some tools such as Sollya[8] (see Section 3.9 and Chapter 4) and Gappa[9] (see Sections 2.2.4 and 5.3) already make it possible to automatically find a good polynomial approximation to some function in a given interval, provide a tight yet certain bound on the approximation error, and bound the error of a polynomial evaluation scheme. The METALIBM project aims at filling the gap toward either full automation (assuming a nonexpert user who wants a sound implementation of a given function) or computer-assisted function design (assuming that the user is an experienced libm designer who wants to quickly obtain a fast and accurate program). Reference [70] presents the state of the project at the time I am writing these lines. Good results are already obtained on architecture-adapted code generation for polynomial evaluation. Range reduction remains a difficult problem: when we have no preliminary knowledge of a specific algebraic property (such as $\exp(x + y) = \exp(x)\exp(y)$ or $\cos(x + 2\pi) = \cos(x)$) that can be used, it seems that splitting the initial domain into subintervals small enough so that a good polynomial approximation can be built in each of them is the only alternative. Progress has been done on this topic [289, 303]. However, this technique cannot be used for very large initial domains, such as the set of the binary64-representable numbers. Breakthroughs in that domain may come from computer algebra. For instance, in his PhD dissertation [344], Mezzarobba shows, for a large class of functions, how we can find asymptotic expansions or detect periodicities just from the coefficients of the differential equation they satisfy. Paper [299] is a very interesting example of what can be done.

[8] http://sollya.gforge.inria.fr
[9] http://gappa.gforge.inria.fr

Bibliography

[1] M. Abramowitz, I.A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*. Applied Math. Series 55 (National Bureau of Standards, Washington, D.C., 1964)

[2] R.C. Agarwal, J.C. Cooley, F.G. Gustavson, J.B. Shearer, G. Slishman, B. Tuckerman, New scalar and vector elementary functions for the IBM system/370. IBM J. Res. Dev. **30**(2), 126–144 (1986)

[3] H.M. Ahmed, Efficient elementary function generation with multipliers, in *Proceedings of the 9th IEEE Symposium on Computer Arithmetic* (1989), pp. 52–59

[4] H.M. Ahmed, J.M. Delosme, M. Morf, Highly concurrent computing structures for matrix arithmetic and signal processing. Computer **15**(1), 65–82 (1982)

[5] H. Alt, Comparison of arithmetic functions with respect to Boolean circuits, in *Proceedings of the 16th ACM STOC* (1984), pp. 466–470

[6] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754–1985 (1985)

[7] C. Ancourt, F. Irigoin, Scanning polyhedra with do loops, in *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'91)*, Apr 1991 (ACM Press, New York, NY, 1991), pp. 39–50

[8] I.J. Anderson, A distillation algorithm for floating-point summation. SIAM J. Sci. Comput. **20**(5), 1797–1806 (1999)

[9] M. Andrews, T. Mraz, Unified elementary function generator. Microprocess. Microsyst. **2**(5), 270–274 (1978)

[10] E. Antelo, J.D. Bruguera, J. Villalba, E. Zapata, Redundant CORDIC rotator based on parallel prediction, in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 172–179

[11] E. Antelo, T. Lang, J.D. Bruguera, Very-high radix CORDIC rotation based on selection by rounding. J. VLSI Signal Process. Syst. **25**(2), 141–154 (2000)

[12] A. Avizienis, Signed-digit number representations for fast parallel arithmetic. IRE Trans. Electron. Comput. **10**, 389–400 (1961). Reprinted in [440]

[13] A. Azmi, F. Lombardi, On a tapered floating point system, in *Proceedings of 9th IEEE Symposium on Computer Arithmetic*, Sept 1989, pp. 2–9

[14] L. Babai, On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**(1), 1–13 (1986)

[15] D.H. Bailey, Algorithm 719, multiprecision translation and execution of FORTRAN programs. ACM Trans. Math. Softw. **19**(3), 288–319 (1993)

[16] D.H. Bailey, Some background on Kanada's recent pi calculation. Technical report, Lawrence Berkeley National Laboratory (2003), http://crd.lbl.gov/~dhbailey/dhbpapers/dhb-kanada.pdf

[17] D.H. Bailey, High-precision floating-point arithmetic in scientific computation. Comput. Sci. Eng. **7**(3), 54–61 (2005)

[18] D.H. Bailey, A thread-safe arbitrary precision computation package (full documentation) (2015), http://www.davidhbailey.com/dhbpapers/index.html#Technical-papers

[19] D.H. Bailey, J.M. Borwein, Experimental mathematics: examples, methods and implications. Not. AMS **52**(5), 502–514 (2005)

[20] D.H. Bailey, J.M. Borwein, High-precision arithmetic in mathematical physics. Mathematics **3**, 337–367 (2015)

[21] D.H. Bailey, J.M. Borwein, P.B. Borwein, S. Plouffe, The quest for pi. Math. Intelligencer **19**(1), 50–57 (1997)

[22] D.H. Bailey, Y. Hida, X.S. Li, B. Thompson, ARPREC: an arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory (2002), http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4

[23] B.L. Bailly, J.P. Thiran, Computing complex polynomial chebyshev approximants on the unit circle by the real remez algorithm. SIAM J. Numer. Anal. **36**(6), 1858–1877 (1999)
[24] J.C. Bajard, J. Duprat, S. Kla, J.M. Muller, Some operators for on-line radix 2 computations. J. Parallel Distrib. Comput. **22**(2), 336–345 (1994)
[25] J.C. Bajard, S. Kla, J.M. Muller, BKM: a new hardware algorithm for complex elementary functions. IEEE Trans. Comput. **43**(8), 955–963 (1994)
[26] G.A. Baker, *Essentials of Padé Approximants* (Academic Press, New York, 1975)
[27] G.A. Baker, *Padé Approximants*. Number 59 in Encyclopedia of Mathematics and its Applications (Cambridge University Press, New York, 1996)
[28] H.G. Baker, Less complex elementary functions. ACM SIGPLAN Not. **27**(11), 15–16 (1992)
[29] P.W. Baker, Suggestion for a fast binary sine/cosine generator. IEEE Trans. Comput. **C-25**(11) (1976)
[30] G.V. Bard, *Sage for Undergraduates* (American Mathematical Society, 2015)
[31] R. Barrio, A. Dena, W. Tucker, A database of rigorous and high-precision periodic orbits of the Lorenz model. Comput. Phys. Commun. **194**, 76–83 (2015)
[32] P. Beame, S. Cook, H. Hoover, Log depth circuits for division and related problems. SIAM J. Comput. **15**, 994–1003 (1986)
[33] M. Bekooij, J. Huisken, K. Nowak, Numerical accuracy of fast fourier transforms with CORDIC arithmetic. J. VLSI Signal Process. Syst. **25**(2), 187–193 (2000)
[34] A. Benoit, F. Chyzak, A. Darrasse, S. Gerhold, M. Mezzarobba, B. Salvy, The dynamic dictionary of mathematical functions (DDMF), in *Mathematical Software—ICMS 2010*. LNCS, vol. 6327, ed. by K. Fukuda, J. van der Hoeven, M. Joswig, N. Takayama (Springer, 2010), pp. 3541
[35] J.-P. Berrut, H. Mittelmann, Adaptive point shifts in rational approximation with optimized denominator. J. Comput. Appl. Math. **164–165**, 81–92 (2004)
[36] C. Bertin, N. Brisebarre, B.D. de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S.-K. Raina, A. Tisserand, A floating-point library for integer processors, in *Proceedings of SPIE 49th Annual Meeting International Symposium on Optical Science and Technology, Denver*. SPIE, Aug 2004
[37] C.M. Black, R.P. Burton, T.H. Miller, The need for an industry standard of accuracy for elementary-function programs. ACM Trans. Math. Softw. **10**(4), 361–366 (1984)
[38] M. Bodrato, Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0, in *WAIFI'07 Proceedings*. LNCS, vol. 4547, ed. by C. Carlet, B. Sunar (Springer, 2007), pp. 116–133
[39] S. Boldo, Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms, in *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. Lecture Notes in Computer Science, vol. 4130, ed. by U. Furbach, N. Shankar (2006), pp. 52–66
[40] S. Boldo, M. Daumas, R.-C. Li, Formally verified argument reduction with a fused multiply-add. IEEE Trans. Comput. **58**(8), 1139–1145 (2009)
[41] S. Boldo, M. Daumas, C. Moreau-Finot, L. Théry, Computer validated proofs of a toolset for adaptable arithmetic. Technical report, École Normale Supérieure de Lyon (2001), http://arxiv.org/pdf/cs.MS/0107025
[42] S. Boldo, M. Daumas, L. Théry, Formal proofs and computations in finite precision arithmetic, in *Proceedings of the 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, ed. by T. Hardin, R. Rioboo (2003)
[43] S. Boldo, J.-C. Filliâtre, Formal verification of floating-point programs, in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic* (2007), pp. 187–194
[44] S. Boldo, G. Melquiond, Emulation of FMA and correctly rounded sums: proved algorithms using rounding to odd. IEEE Trans. Comput. **57**(4), 462–471 (2008)
[45] S. Boldo, J.-M. Muller, Some functions computable with a fused-mac, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, June 2005, pp. 52–58
[46] S. Boldo, J.-M. Muller, Exact and approximated error of the FMA. IEEE Trans. Comput. **60**, 157–164 (2011)
[47] A.D. Booth, A signed binary multiplication technique. Q. J. Mech. Appl. Math. **4**(2), 236–240 (1951). Reprinted in [439]
[48] F. Bornemann, D. Laurie, S. Wagon, J. Waldvogel, *The SIAM 100-Digit Challenge* (SIAM, 2004)
[49] J. Borwein, D.H. Bailey, *Mathematics by Experiment: Plausible Reasoning in the 21st Century* (A. K. Peters, Natick, MA, 2004)
[50] J.M. Borwein, P.B. Borwein, The arithmetic-geometric mean and fast computation of elementary functions. SIAM Rev. **26**(3), 351–366 (1984)
[51] J.M. Borwein, P.B. Borwein, On the complexity of familiar functions and numbers. SIAM Rev. **30**(4), 589–601 (1988)
[52] P. Borwein, T. Erdélyi, *Polynomials and Polynomial Inequalities*. Graduate Texts in Mathematics, vol. 161 (Springer, New York, 1995)
[53] E.L. Braun, *Digital Computer Design* (Academic Press, New York, 1963)

[54] K. Braune, Standard functions for real and complex point and interval arguments with dynamic accuracy. Comput. Suppl. **6**, 159–184 (1988)
[55] R.P. Brent, On the precision attainable with various floating point number systems. IEEE Trans. Comput. **C-22**(6), 601–607 (1973)
[56] R.P. Brent, Multiple precision zero-finding methods and the complexity of elementary function evaluation, in *Analytic Computational Complexity*, ed. by J.F. Traub (Academic Press, New York, 1975)
[57] R.P. Brent, Fast multiple precision evaluation of elementary functions. J. ACM **23**, 242–251 (1976)
[58] R.P. Brent, Algorithm 524, MP, a FORTRAN multiple-precision arithmetic package. ACM Trans. Math. Softw. **4**(1), 71–81 (1978)
[59] R.P. Brent, A FORTRAN multiple-precision arithmetic package. ACM Trans. Math. Softw. **4**(1), 57–70 (1978)
[60] R.P. Brent, Unrestricted algorithms for elementary and special functions, in *Information Processing 80*, ed. by S.H. Lavington (North-Holland, Amsterdam, 1980), pp. 613–619
[61] R.P. Brent, P. Zimmermann, *Modern Computer Arithmetic*. Cambridge Monographs on Applied and Computational Mathematics, vol. 18 (Cambridge University Press, 2010)
[62] W.S. Briggs, D.W. Matula, A 17 x 69 bit multiply and add unit with redundant binary feedback and single cycle latency, in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, June 1993, pp. 163–171. Reprinted in [442]
[63] N. Brisebarre, S. Chevillard, Efficient polynomial L^∞ approximations, in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic* (2007), pp. 169–176
[64] N. Brisebarre, D. Defour, P. Kornerup, J.-M. Muller, N. Revol, A new range reduction algorithm. IEEE Trans. Comput. **54**(3), 331–339 (2005)
[65] N. Brisebarre, M.D. Ercegovac, J.-M. Muller, (m, p, k)-friendly points: a table-based method for trigonometric function evaluation, in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, Los Alamitos, CA, USA, July 2012, pp. 46–52. IEEE Computer Society
[66] N. Brisebarre, J.-M. Muller, Correct rounding of algebraic functions. Theor. Inf. Appl. **41**, 71–83 (2007)
[67] N. Brisebarre, J.-M. Muller, Correctly rounded multiplication by arbitrary precision constants. IEEE Trans. Comput. **57**(2), 165–174 (2008)
[68] N. Brisebarre, J.-M. Muller, S.-K. Raina, Accelerating correctly rounded floating-point division when the divisor is known in advance. IEEE Trans. Comput. **53**(8), 1069–1072 (2004)
[69] N. Brisebarre, J.-M. Muller, A. Tisserand, Computing machine-efficient polynomial approximations. Draft, LIP Laboratory (2004), http://perso.ens-lyon.fr/jean-michel.muller/bmt-toms.ps
[70] N. Brunie, F. de Dinechin, O. Kupriianova, C. Lauter, Code generators for mathematical functions, in *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, June 2015, pp. 66–73
[71] A. Bultheel, P. Gonzales-Vera, E. Hendriksen, O. Njastad, *Orthogonal Rational Functions*. Cambridge Monographs on Applied and Computational Mathematics, vol. 5 (Cambridge University Press, New York, 1999)
[72] J.W. Carr III, A.J. Perlis, J.E. Robertson, N.R. Scott, A visit to computation centers in the Soviet Union. Commun. ACM **2**(6), 8–20 (1959)
[73] A. Cauchy, Sur les moyens d'éviter les erreurs dans les calculs numériques. *Comptes Rendus de l'Académie des Sciences, Paris*, 11:789–798 (1840). Republished in: Augustin Cauchy, oeuvres complètes, 1ère série, Tome V, pp. 431–442, http://gallica.bnf.fr/scripts/ConsultationTout.exe?O=N090185
[74] J.R. Cavallaro, N.D. Hemkumar, Efficient complex matrix transformations with CORDIC, in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, June 1993, pp. 122–129
[75] J.R. Cavallaro, F.T. Luk, CORDIC arithmetic for an SVD processor, in *Proceedings of the 8th IEEE Symposium on Computer Arithmetic* (1988), pp. 113–120
[76] J.R. Cavallaro, F.T. Luk, Floating-point CORDIC for matrix computations, in *Proceedings of the 1988 IEEE International Conference on Computer Design* (1988), pp. 40–42
[77] L.W. Chang, S.W. Lee, Systolic arrays for the discrete Hartley transform. IEEE Trans. Signal Process. **39**(11), 2411–2418 (1991)
[78] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, S.M. Watt, *Maple V Library Reference Manual* (Springer, Berlin, 1991)
[79] T.C. Chen, Automatic computation of logarithms, exponentials, ratios and square roots. IBM J. Res. Dev. **16**, 380–388 (1972)
[80] E.W. Cheney, *Introduction to Approximation Theory*, International Series in Pure and Applied Mathematics (McGraw Hill, New York, 1966)
[81] E.W. Cheney, *Introduction to Approximation Theory*, 2nd edn. (AMS Chelsea Publishing, Providence, RI, 1982)
[82] S.H. Cheng, N.J. Higham, C.S. Kenney, A.J. Laub, Approximating the logarithm of a matrix to specified accuracy. SIAM J. Matrix Anal. Appl. **22**(4), 1112–1125 (2001)
[83] S. Chevillard, J. Harrison, M. Joldes, C. Lauter, Efficient and accurate computation of upper bounds of approximation errors. Theoret. Comput. Sci. **412**(16), 1523–1543 (2011)

[84] S. Chevillard, J. Harrison, M.M. Joldes, C. Lauter, Efficient and accurate computation of upper bounds of approximation errors. J. Theoret. Comput. Sci. **412**(16), 1523–1543 (2011)
[85] S. Chevillard, M. Joldeş, C. Lauter, Sollya: an environment for the development of numerical codes, in *Mathematical Software—ICMS 2010*, vol. 6327, Lecture Notes in Computer Science, ed. by K. Fukuda, J. van der Hoeven, M. Joswig, N. Takayama (Springer, Heidelberg, 2010), pp. 28–31
[86] S. Chevillard, C.Q. Lauter, A certified infinite norm for the implementation of elementary functions, in *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE (2007), pp. 153–160
[87] S. Chevillard, M. Mezzarobba, Multiple precision evaluation of the Airy Ai function with reduced cancellation, in *Proceedings of the 21st IEEE Symposium on Computer Arithmetic* (2013), pp. 175–182
[88] C.Y. Chow, J.E. Robertson, Logical design of a redundant binary adder, in *Proceedings of the 4th IEEE Symposium on Computer Arithmetic* (1978)
[89] D.V. Chudnovsky, G.V. Chudnovsky, The computation of classical constants. Proc. Nat. Acad. Sci. **86**(21), 8178–8182 (1989)
[90] C.W. Clenshaw, Rational approximations for special functions, in *Software for Numerical Mathematics*, ed. by D.J. Evans (Academic Press, New York, 1974)
[91] C.W. Clenshaw, F.W.J. Olver, Beyond floating point. J. ACM **31**, 319–328 (1985)
[92] D. Cochran, Algorithms and accuracy in the HP 35. Hewlett Packard J. **23**, 10–11 (1972)
[93] W. Cody, W. Waite, *Software Manual for the Elementary Functions* (Prentice-Hall, Englewood Cliffs, 1980)
[94] W.J. Cody, A survey of practical rational and polynomial approximation of functions. SIAM Rev. **12**(3), 400–423 (1970)
[95] W.J. Cody, Static and dynamic numerical characteristics of floating-point arithmetic. IEEE Trans. Comput. **C-22**(6), 598–601 (1973)
[96] W.J. Cody, Funpack, a package of special function subroutines. Technical Memorandum 385, Argonne National Laboratory, Argonne, IL (1981)
[97] W.J. Cody, Implementation and testing of function software, in *Problems and Methodologies in Mathematical Software Production*, vol. 142, Lecture Notes in Computer Science, ed. by P.C. Messina, A. Murli (Springer, Berlin, 1982)
[98] W.J. Cody, MACHAR: a subroutine to dynamically determine machine parameters. ACM Trans. Math. Softw. **14**(4), 301–311 (1988)
[99] W.J. Cody, Performance evaluation of programs for the error and complementary error functions. ACM Trans. Math. Softw. **16**(1), 29–37 (1990)
[100] W.J. Cody, Algorithm 715: SPECFUN—a portable FORTRAN package for special function routines and test drivers. ACM Trans. Math. Softw. **19**(1), 22–32 (1993)
[101] W.J. Cody, CELEFUNT: a portable test package for complex elementary functions. ACM Trans. Math. Softw. **19**(1), 1–21 (1993)
[102] W.J. Cody, J.T. Coonen, Algorithm 722: functions to support the IEEE standard for binary floating-point arithmetic. ACM Trans. Math. Softw. **19**(4), 443–451 (1993)
[103] W.J. Cody, J.T. Coonen, D.M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F.N. Ris, D. Stevenson, A proposed radix-and-word-length-independent standard for floating-point arithmetic. IEEE MICRO **4**(4), 86–100 (1984)
[104] W.J. Cody, L. Stoltz, The use of taylor series to test accuracy of function programs. ACM Trans. Math. Softw. **17**(1), 55–63 (1991)
[105] J.-F. Collard, P. Feautrier, T. Risset, Construction of do loops from systems of affine constraints. Parallel Process. Lett. **5**, 421–436 (1995)
[106] V. Considine, CORDIC trigonometric function generator for DSP, in *Proceedings of 1989 International Conference on Acoustics, Speech and Signal Processing* (1989), pp. 2381–2384
[107] S.A. Cook, *On the minimum computation time of functions*. PhD thesis, Department of Mathematics, Harvard University (1966)
[108] J.W. Cooley, J.W. Tukey, An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**(90), 297–301 (1965)
[109] J.T. Coonen, An implementation guide to a proposed standard for floating-point arithmetic. *Computer* (1980)
[110] D. Coppersmith, Finding a small root of a univariate modular equation, in *Proceedings of EUROCRYPT*, vol. 1070, Lecture Notes in Computer Science, ed. by U.M. Maurer (Springer, Berlin, 1996), pp. 155–165
[111] D. Coppersmith, Finding small solutions to small degree polynomials, in *Proceedings of Cryptography and Lattices (CaLC)*, vol. 2146, Lecture Notes in Computer Science, ed. by J.H. Silverman (Springer, Berlin, 2001), pp. 20–31
[112] R.M. Corless, D.J. Jeffrey, S.M. Watt, J.H. Davenport, "According to Abramowitz and Stegun" or arccoth needn't be uncouth. SIGSAM Bull. **34**(2), 58–65 (2000)

[113] M. Cornea, R.A. Golliver, P. Markstein, Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms, in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Apr 1999, pp. 96–105
[114] M. Cornea, J. Harrison, C. Anderson, P.T.P. Tang, E. Schneider, E. Gvozdev, A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. IEEE Trans. Comput. **58**(2), 148–162 (2009)
[115] M. Cornea, J. Harrison, P.T.P. Tang, *Scientific Computing on Itanium ® -based Systems* (Intel Press, Hillsboro, OR, 2002)
[116] M. Cosnard, A. Guyot, B. Hochet, J.M. Muller, H. Ouaouicha, P. Paul, E. Zysman, The FELIN arithmetic coprocessor chip, in *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, May 1987
[117] M.F. Cowlishaw, Decimal floating-point: algorism for computers, in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 104–111
[118] A.A. Cuyt, V. Petersen, B. Verdonk, H. Waadeland, W.B. Jones, *Handbook of Continued Fractions for Special Functions*, 1st edn. (Springer Publishing Company, Incorporated, 2008)
[119] T. Cyrix Corporation, Richardson. *FastMath Accuracy*, Report (1989)
[120] T. Cyrix Corporation, Richardson. *Cyrix 6x86 Processor Data Book* (1996)
[121] L. Dadda, Some schemes for parallel multipliers. Alta Frequenza **34**, 349–356 (1965). Reprinted in [439]
[122] D.H. Daggett, Decimal-binary conversion in CORDIC. IRE Trans. Electron. Comput. **EC-8**(3), 335–339 (1959)
[123] A. Dahan-Dalmedico, J. Pfeiffer, *Histoire des Mathématiques* (Editions du Seuil, Paris, 1986). In French
[124] C. Daramy, D. Defour, F. de Dinechin, J.-M. Muller, CR-LIBM, a correctly rounded elementary function library, in *SPIE 48th Annual Meeting International Symposium on Optical Science and Technology*, Aug 2003
[125] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C.Q. Lauter, J.-M. Muller, CR-LIBM, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arenaire team, https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf, Dec 2006
[126] M. Daumas, C. Mazenc, X. Merrheim, J.-M. Muller, Fast and accurate range reduction for computation of the elementary functions, in *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*. IMACS, Piscataway, NJ (1994), pp. 1196–1198
[127] M. Daumas, C. Mazenc, X. Merrheim, J.-M. Muller, Modular range reduction: a new algorithm for fast and accurate computation of the elementary functions. J. Univ. Comput. Sci. **1**(3), 162–175 (1995)
[128] M. Daumas, G. Melquiond, Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**(1), 2:1–2:20 (2010)
[129] M. Daumas, G. Melquiond, C. Muñoz, Guaranteed proofs using interval arithmetic, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (2005), pp. 188–195
[130] H. Dawid, H. Meyr, The differential CORDIC algorithm: constant scale factor redundant implementation without correcting iterations. IEEE Trans. Comput. **45**(3), 307–318 (1996)
[131] F. de Dinechin, A.V. Ershov, N. Gast, Towards the post-ultimate libm, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (2005), pp. 288–295
[132] F. de Dinechin, N. Gast, Towards the post-ultimate libm. Research Report 2004-47, LIP, École normale supérieure de Lyon (2004), http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-47.pdf
[133] F. de Dinechin, C. Lauter, G. Melquiond, Assisted verification of elementary functions using Gappa, in *Proceedings of the 2006 ACM Symposium on Applied Computing*, Dijon, France (2006), pp. 1318–1322
[134] F. de Dinechin, C. Lauter, G. Melquiond, Certifying the floating-point implementation of an elementary function using Gappa. Trans. Comput. **60**(2), 242–253 (2011)
[135] F. de Dinechin, C. Lauter, J.-M. Muller, S. Torres, On Ziv's rounding test. ACM Trans. Math. Softw. **39**(4) (2013)
[136] F. de Dinechin, C.Q. Lauter, J.-M. Muller, Fast and correctly rounded logarithms in double-precision. Theor. Inf. Appl. **41**, 85–102 (2007)
[137] F. de Dinechin, B. Pasca, Designing custom arithmetic data paths with FloPoCo. IEEE Des. Test Comput. **28**(4), 18–27 (2011)
[138] F. de Dinechin, A. Tisserand, Some improvements on multipartite table methods, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (2001), pp. 128–135. Reprinted in [442]
[139] F. de Dinechin, A. Tisserand, Multipartite table methods. IEEE Trans. Comput. **54**(3), 319–330 (2005)
[140] C.J. de La Vallée Poussin, *L'approximation des Fonctions d'une Variable Réelle (in French)* (Gauthier-Villars, Paris, 1919)
[141] H. de Lassus Saint-Genies, D. Defour, G. Revy, Range reduction based on pythagorean triples for trigonometric function evaluation, in *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 74–81
[142] G. Deaconu, C. Louembet, A. Theron, Designing continuously constrained spacecraft relative trajectories for proximity operations. J. Guidance Control Dyn. **38**(7), 1208–1217 (2015)
[143] D. Defour, Cache-optimised methods for the evaluation of elementary functions. Technical Report RR2002-38, LIP Laboratory, ENS Lyon, ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2002/RR2002-38.ps.gz, Oct 2002

[144] D. Defour, *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision (in French)*. PhD thesis, École Normale Supérieure de Lyon, Sept 2003
[145] D. Defour, F. de Dinechin, Software carry-save for fast multiple-precision algorithms, in *35th International Congress of Mathematical Software*, Aug 2002, pp. 29–40
[146] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, P. Zimmermann, Proposal for a standardization of mathematical function implementation in floating-point arithmetic. Numer. Algorithms **37**(1–4), 367–375 (2004)
[147] D. Defour, P. Kornerup, J.-M. Muller, N. Revol, A new range reduction algorithm, in *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers*, vol. 2 (2001), pp. 1656–1660
[148] T.J. Dekker, A floating-point technique for extending the available precision. Numer. Math. **18**(3), 224–242 (1971)
[149] J.M. Delosme, A processor for two-dimensional symmetric eigenvalue and singular value arrays, in *Twenty-First Asilomar Conference on Circuits, Systems, and Computers*, Nov 1987, pp. 217–221
[150] J.M. Delosme, Bit-level systolic algorithms for real symmetric and hermitian eigenvalue problems. J. VLSI Signal Process. **4**, 69–88 (1992)
[151] B. DeLugish, *A class of algorithms for automatic evaluation of functions and computations in a digital computer*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL (1970)
[152] E. Deprettere, P. Dewilde, R. Udo, Pipelined CORDIC architectures for fast VLSI filtering and array processing, in *Proceedings of ICASSP'84* (1984), pp. 41.A.6.1–41.A.6.4
[153] E.F. Deprettere, A.J. de Lange, Design and implementation of a floating-point quasi-systolic general purpose CORDIC rotator for high-rate parallel data and signal processing, in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 272–281
[154] A.M. Despain, Fourier transform computers using CORDIC iterations. IEEE Trans. Comput. **C-33**(5) (1974)
[155] P. Deuflhard, A short history of Newton's method. Doc. Math. **ISMP**, 25–30 (2012)
[156] J.V. Deun, A. Bultheel, An interpolation algorithm for orthogonal rational functions. J. Comput. Appl. Math. **164–165**, 749–762 (2004)
[157] L. Didier, F. Rico, High radix BKM algorithm. Numer. Algorithms **37**(1–4), 113–125 (2004)
[158] W.S. Dorn, Generalizations of Horner's rule for polynomial evaluation. IBM J. Res. Dev. **6**(2), 239–245 (1962)
[159] C.B. Dunham, Rational approximation with a vanishing weight function and with a fixed value at zero. Math. Comput. **30**(133), 45–47 (1976)
[160] C.B. Dunham, Choice of basis for Chebyshev approximation. ACM Trans. Math. Softw. **8**(1), 21–25 (1982)
[161] C.B. Dunham, Provably monotone approximations I. SIGNUM Newsl. **22**, 6–11 (1987)
[162] C.B. Dunham, Provably monotone approximations, II. SIGNUM Newsl. **22**, 30–31 (1987)
[163] C.B. Dunham, Feasibility of "perfect" function evaluation. SIGNUM Newsl. **25**(4), 25–26 (1990)
[164] C.B. Dunham, Fitting approximations to the Kuki-Cody-Waite form. Int. J. Comput. Math. **31**, 263–265 (1990)
[165] C.B. Dunham, Provably monotone approximations, IV. Technical Report 422, Department of Computer Science, The University of Western Ontario, London, Canada (1994)
[166] C.B. Dunham, Approximation with taylor matching at the origin. Int. J. Comput. Math. **80**(8), 1019–1024 (2003)
[167] J. Duprat, J.-M. Muller, Hardwired polynomial evaluation. J. Parallel Distrib. Comput., Special Issue on Parallelism in Computer Arithmetic (5) (1988)
[168] J. Duprat, J.-M. Muller, The CORDIC algorithm: new results for fast VLSI implementation. IEEE Trans. Comput. **42**(2), 168–178 (1993)
[169] S.W. Ellacott, On the Faber transform and efficient numerical rational approximation. SIAM J. Numer. Anal. **20**(5), 989–1000 (1983)
[170] M. Ercegovac, T. Lang, J.-M. Muller, A. Tisserand, Reciprocation, square root, inverse square root and some elementary functions using small multipliers. IEEE Trans. Comput. **49**(7), 628–637 (2000). Reprinted in [442]
[171] M.D. Ercegovac, Radix 16 evaluation of certain elementary functions. IEEE Trans. Comput. **C-22**(6), 561–566, June 1973. Reprinted in [439]
[172] M.D. Ercegovac, *A general method for evaluation of functions and computation in a digital computer*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL (1975)
[173] M.D. Ercegovac, A general hardware-oriented method for evaluation of functions and computations in a digital computer. IEEE Trans. Comput. **C-26**(7), 667–680 (1977)
[174] M.D. Ercegovac, On-line arithmetic: an overview, in *SPIE, Real Time Signal Processing VII*. SPIE-The International Society for Optical Engeneering, Bellingham, WA (1984), pp. 86–93
[175] M.D. Ercegovac, T. Lang, Fast cosine/sine implementation using on-line CORDIC, in *Twenty-First Asilomar Conference on Signals, Systems, and Computers* (1987)
[176] M.D. Ercegovac, T. Lang, On-the-fly conversion of redundant into conventional representations. IEEE Trans. Comput. **C-36**(7), 895–897 (1987). Reprinted in [440]
[177] M.D. Ercegovac, T. Lang, On-line scheme for computing rotation factors. J. Parallel Distrib. Comput. Special Issue on Parallelism in Computer Arithmetic (5), 209–227 (1988). Reprinted in [440]
[178] M.D. Ercegovac, T. Lang, Redundant and on-line CORDIC: application to matrix triangularization and SVD. IEEE Trans. Comput. **39**(6), 725–740 (1990)

[179] M.D. Ercegovac, T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations* (Kluwer Academic Publishers, Boston, 1994)
[180] M.D. Ercegovac, T. Lang, *Digital Arithmetic* (Morgan Kaufmann Publishers, San Francisco, 2004)
[181] M.D. Ercegovac, T. Lang, P. Montuschi, Very-high radix division with prescaling and selection by rounding. IEEE Trans. Comput. **43**(8), 909–918 (1994)
[182] M.D. Ercegovac, K.S. Trivedi, On-line algorithms for division and multiplication. IEEE Trans. Comput. **C-26**(7), 681–687 (1977). Reprinted in [440]
[183] M.A. Erle, M.J. Schulte, B.J. Hickmann, Decimal floating-point multiplication via carry-save addition, in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, June 2007, pp. 46–55
[184] G. Estrin, Organization of computer systems—the fixed plus variable structure computer, in *Proceedings Western Joint Computing Conference*, vol. 17, pp. 33–40 (1960)
[185] A. Feldstein, R. Goodman, Convergence estimates for the distribution of trailing digits. J. ACM **23**, 287–297 (1976)
[186] W. Ferguson, Exact computation of a sum or difference with applications to argument reduction, in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 216–221
[187] W. Ferguson, Private communication. Unpublished (1997)
[188] W. Ferguson, T. Brightman, Accurate and monotone approximations of some transcendental functions, in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 237–244. Reprinted in [442]
[189] C.T. Fike, Methods for evaluating polynomial approximations in function evaluation routines. Commun. ACM **10**(3), 175–178 (1967)
[190] B.P. Flannery, W.H. Press, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes in C*, 2nd edn. (Cambridge University Press, New York, 1992)
[191] M.J. Flynn, S.F. Oberman, *Advanced Computer Arithmetic Design* (Wiley-Interscience, 2001)
[192] A. Fog, The microarchitecture of Intel, AMD and VIA CPUs: an optimization guide for assembly programmers and compiler makers. Technical report, Technical University of Denmark (2014), http://www.agner.org/optimize/
[193] P. Fortin, M. Gouicem, S. Graillat, Towards solving the table maker's dilemma on GPU, in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2012, pp. 407–415
[194] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2) (2007), http://www.mpfr.org/
[195] D. Fowler, E. Robson, Square root approximations in old Babylonian mathematics: YBC 7289 in context. Historia Mathematica **25**, 366–378 (1998)
[196] W. Fraser, A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. J. ACM **12**(3), 295–314 (1965)
[197] M. Fürer, Faster integer multiplication, in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, CA*, ed. by D.S. Johnson, U. Feige, June 2007. ACM, pp. 57–66
[198] S. Gal, Computing elementary functions: a new approach for achieving high accuracy and good performance, *Accurate Scientific Computations*, vol. 235, Lecture Notes in Computer Science (Springer, Berlin, 1986), pp. 1–16
[199] S. Gal, B. Bachelis, An accurate elementary mathematical library for the IEEE floating point standard. ACM Trans. Math. Softw. **17**(1), 26–45 (1991)
[200] M. Garrido, J. Grajal, Efficient memoryless CORDIC for FFT computation, in *IEEE International Conference on Acoustics, Speech and Signal Processing, 2007*, vol. 2, April 2007, pp. II–113–II–116
[201] M. Garrido, P. Kallstrom, M. Kumm, O. Gustafsson, CORDIC II: a new improved CORDIC algorithm. IEEE Trans. Circuits Syst. II: express briefs **63**(2), 186–190 (2016)
[202] W. Gautschi, *Numerical Analysis: An Introduction* (Birkhäuser, Boston, 1997)
[203] W. Gautschi, G.H. Golub, G. Opfer (eds.), *Applications and Computation of Orthogonal Polynomials* (International Series of Numerical Mathematics. Birkhäuser, Basel, 1999)
[204] W.M. Gentleman, S.B. Marovitch, More on algorithms that reveal properties of floating-point arithmetic units. Commun. ACM **17**(5), 276–277 (1974)
[205] A. Gil, J. Segura, N. Temme, *Numerical Methods for Special Functions*. Society for Industrial and Applied Mathematics (2007)
[206] D. Goldberg, What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–47, Mar 1991. An edited reprint is available at http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, http://www.validlab.com/goldberg/addendum.html
[207] X. Gourdon, P. Sebah, Binary splitting methods (2001), http://numbers.computation.free.fr/Constants/Algorithms/splitting.ps
[208] P.J. Grabner, C. Heuberger, On the number of optimal base 2 representations of integers. Des. Codes Crypt. **40**, 25–39 (2006)
[209] S. Graillat, V. Lefèvre, J.-M. Muller, On the maximum relative error when computing integer powers by iterated multiplications in floating-point arithmetic. *Numerical Algorithms* (2015), pp. 1–15

[210] T. Granlund, The GNU multiple precision arithmetic library, release 4.1.4. Sept 2004, http://gmplib.org/gmp-man-4.1.4.pdf
[211] J. Gustafson, *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science (Taylor & Francis, 2015)
[212] B. Haible, T. Papanikolaou, Fast multiprecision evaluation of series of rational numbers, in *Algorithmic Number Theory*, vol. 1423, Lecture Notes in Computer Science, ed. by J. Buhler (Springer, Berlin, 1998), pp. 338–350
[213] H. Hamada, A new approximation form for mathematical functions, in *Proceedings of SCAN-95, IMACS/GAMM Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Sept 1995
[214] E.R. Hansen, M.L. Patrick, R.L.C. Wang, Polynomial evaluation with scaling. ACM Trans. Math. Softw. **16**(1), 86–93 (1990)
[215] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, N. Takagi, A high-speed multiplier using a redundant binary adder tree. IEEE J. Solid-State Circuits **SC-22**(1), 28–34 (1987). Reprinted in [440]
[216] J. Harrison, Floating-point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory (1997)
[217] J. Harrison, A machine-checked theory of floating-point arithmetic, in *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, Lecture Notes in Computer Science, vol. 1690, ed. by Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (Springer, Berlin, 1999), pp. 113–130
[218] J. Harrison, Formal verification of floating-point trigonometric functions, in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000*, number 1954 in Lecture Notes in Computer Science, ed. by W.A. Hunt, S.D. Johnson (Springer, Berlin, 2000), pp. 217–233
[219] J. Harrison, Formal verification of IA-64 division algorithms, in *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2000*. Lecture Notes in Computer Science, vol. 1869, ed. by M. Aagaard, J. Harrison (Springer, 2000), pp. 234–251
[220] J. Harrison, Floating-point verification using theorem proving, in *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*. Lecture Notes in Computer Science, vol. 3965, ed. by M. Bernardo, A. Cimatti (Springer, Bertinoro, Italy, 2006), pp. 211–242
[221] J. Harrison, Verifying nonlinear real formulas via sums of squares, in *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*. Lecture Notes in Computer Science, vol. 4732, ed. by K. Schneider, J. Brandt (Springer, Kaiserslautern, Germany, 2007), pp. 102–118
[222] J. Harrison, Decimal transcendentals via binary, in *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, June 2009, pp. 187–194
[223] J. Harrison, Fast and accurate Bessel function computation, in *Proceedings of the 19th IEEE Symposium on Computer Arithmetic* (2009), pp. 104–113
[224] J. Harrison, T. Kubaska, S. Story, P.T.P. Tang, The computation of transcendental functions on the IA-64 architecture. Intel Technol. J. Q4 (1999), http://developer.intel.com/technology/itj/archive/1999.htm
[225] J.F. Hart, E.W. Cheney, C.L. Lawson, H.J. Maehly, C.K. Mesztenyi, J.R. Rice, H.G. Thacher, C. Witzgall, *Computer Approximations* (Wiley, New York, 1968)
[226] D. Harvey, J.v.d. Hoeven, G. Lecerf, Even faster integer multiplication. Technical report, ArXiv (2014), http://arxiv.org/abs/1407.3360
[227] J.R. Hauser, Handling floating-point exceptions in numeric programs. Technical Report UCB//CSD-95-870, Computer Science Division, University of California, Berkeley, CA, Mar 1995
[228] G.H. Haviland, A.A. Tuszinsky, A CORDIC arithmetic processor chip. IEEE Trans. Comput. **C-29**(2) (1980)
[229] G.H. Hekstra, E.F.A. Deprettere, Floating-point CORDIC, in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, June 1993, pp. 130–137
[230] N.D. Hemkumar, J.R. Cavallaro, Redundant and on-line CORDIC for unitary transformations. IEEE Trans. Comput. **43**(8), 941–954 (1994)
[231] Y. Hida, X.S. Li, D.H. Bailey, Algorithms for quad-double precision floating-point arithmetic, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, June 2001, pp. 155–162
[232] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edn. (SIAM, Philadelphia, 2002)
[233] N.J. Higham, *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics (Philadelphia, PA, USA, 2008)
[234] N.J. Higham, M.I. Smith, Computing the matrix cosine. Numer. Algorithms **34**, 13–26 (2003)
[235] E. Hokenek, R.K. Montoye, P.W. Cook, Second-generation RISC floating point with multiply-add fused. IEEE J. Solid-State Circuits **25**(5), 1207–1213 (1990)
[236] W.G. Horner, A new method of solving numerical equations of all orders by continuous approximation. Philos. Trans. R. Soc. Lond. **109**, 308–335 (1819), http://www.jstor.org/stable/107508
[237] S. Hsiao, C. Lau, J.-M. Delosme, Redundant constant-factor implementation of multi-dimensional CORDIC and its application to complex SVD. J. VLSI Signal Process. Syst. **25**(2), 155–166 (2000)
[238] S.F. Hsiao, J.M. Delosme, Householder CORDIC algorithms. IEEE Trans. Comput. **44**(8), 990–1000 (1995)

[239] X. Hu, S.C. Bass, R.G. Harber, An efficient implementation of singular value decomposition rotation transformations with CORDIC processors. J. Parallel Distrib. Comput. **17**, 360–362 (1993)
[240] Y.H. Hu, The quantization effects of the CORDIC algorithm. IEEE Trans. Signal Process. **40**(4), 834–844 (1992)
[241] Y.H. Hu, S. Naganathan, An angle recoding method for CORDIC algorithm implementation. IEEE Trans. Comput. **42**(1), 99–102 (1993)
[242] T.E. Hull, T.F. Fairgrieve, P.T.P. Tang, Implementing complex elementary functions using exception handling. ACM Trans. Math. Softw. **20**(2), 215–244 (1994)
[243] T.E. Hull, T.F. Fairgrieve, P.T.P. Tang, Implementing the complex arcsine and arccosine functions using exception handling. ACM Trans. Math. Softw. **23**(3), 299–335 (1997)
[244] K. Hwang, *Computer Arithmetic Principles*, Architecture and design (Wiley, New York, 1979)
[245] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug 2008, http://ieeexplore.ieee.org/servlet/opac?punumber=4610933
[246] L. Imbert, J. Muller, F. Rico, Radix-10 BKM algorithm for computing transcendentals on a pocket computer. J. VLSI Signal Process. **25**(2), 179–186 (2000)
[247] International Organization for Standardization, Information technology— Language independent arithmetic— Part 2: Elementary numerical functions. ISO/IEC standard 10967–2 (2001)
[248] C. Iordache, D.W. Matula, On infinitely precise rounding for division, square root, reciprocal and square root reciprocal, in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Apr 1999, pp. 233–240
[249] A. Iserles, A. Zanna, Efficient computation of the matrix exponential by generalized polar decompositions. SIAM J. Numer. Anal. **42**(5), 2218–2256 (2005)
[250] F. Jaime, M. Sanchez, J. Hormigo, J. Villalba, E. Zapata, High-speed algorithms and architectures for range reductioncomputation. IEEE Trans. Very Large Scale Integ. (VLSI) Syst. **19**(3), 512–516 (2011)
[251] F. Jaime, J. Villalba, J. Hormigo, E. Zapata, Pipelined range reduction for floating point numbers, in *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, Sept 2006, pp. 145–152
[252] F.J. Jaime, M.A. Sánchez, J. Hormigo, J. Villalba, E.L. Zapata, Enhanced scaling-free CORDIC. IEEE Trans. Circuits Syst. Part I **57**(7), 1654–1662 (2010)
[253] C.-P. Jeannerod, N. Louvet, J.-M. Muller, Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants. Math. Comput. **82**(284), 2245–2264 (2013)
[254] C.-P. Jeannerod, N. Louvet, J.-M. Muller, A. Panhaleux, Midpoints and exact points of some algebraic functions in floating-point arithmetic. IEEE Trans. Comput. **60**(2) (2011)
[255] C.-P. Jeannerod, C. Mouilleron, J.-M. Muller, G. Revy, C. Bertin, J. Jourdan-Lu, H. Knochel, C. Monat, Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors, in *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, New York, NY, USA (ACM, 2010), pp. 1–9
[256] R.M. Jessani, C.H. Olson, The floating-point unit of the PowerPC 603e microprocessor. IBM J. Res. Dev. **40**(5), 559–566 (1996)
[257] F. Johansson, Evaluating parametric holonomic sequences using rectangular splitting, in *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, New York, NY, USA (ACM, 2014), pp. 256–263
[258] F. Johansson, Efficient implementation of elementary functions in the medium-precision range, in *Proceedings of the 22nd Symposium on Computer Arithmetic*, June 2015, pp. 83–89
[259] M. Joldes, *Rigorous polynomial approximations and applications*. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France (2011)
[260] M. Joldes, V. Popescu, W. Tucker, Searching for sinks for the Henon map using a multiple-precision GPU arithmetic library. SIGARCH Comput. Archit. News **42**(4), 63–68 (2014)
[261] W. Kahan, Pracniques: further remarks on reducing truncation errors. Commun. ACM **8**(1), 40 (1965)
[262] W. Kahan, Minimizing q*m-n. Text accessible electronically at http://http.cs.berkeley.edu/~wkahan/. At the beginning of the file "nearpi.c" (1983)
[263] W. Kahan, Branch cuts for complex elementary functions, in *The State of the Art in Numerical Analysis*, ed. by A. Iserles, M.J.D. Powell (Clarendon Press, Oxford, 1987), pp. 165–211
[264] W. Kahan, Paradoxes in concepts of accuracy, in *Lecture notes from Joint Seminar on Issues and Directions in Scientific Computations, U.C. Berkeley* (1989)
[265] W. Kahan, Lecture notes on the status of IEEE-754. PDF file accessible at http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF (1996)
[266] W. Kahan, IEEE 754: an interview with William Kahan. Computer **31**(3), 114–115 (1998)
[267] W. Kahan, A logarithm too clever by half (2004), http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT
[268] A. Karatsuba, Y. Ofman, Multiplication of many-digital numbers by automatic computers. Doklady Akad. Nauk SSSR **145**, 293–294 (1962). Translation in Physics-Doklady **7**(595–596) (1963)

[269] A.H. Karp, P. Markstein, High-precision division and square root. ACM Trans. Math. Softw. **23**(4), 561–589 (1997)
[270] R. Karpinsky, PARANOIA: a floating-point benchmark. BYTE **10**(2) (1985)
[271] A.Y. Khinchin, *Continued Fractions* (Dover, New York, 1997)
[272] N.G. Kingsbury, P.J.W. Rayner, Digital filtering using logarithmic arithmetic. Electron. Lett. **7**, 56–58 (1971). Reprinted in [439]
[273] P. Kirchberger, *Ueber Tchebychefsche Annaeherungsmethoden*. Ph.D. thesis, Gottingen (1902)
[274] A. Klein, A generalized Kahan-Babuška-summation-algorithm. Computing **76**, 279–293 (2006)
[275] D. Knuth, *The Art of Computer Programming*, vol. 2, 3rd edn. (Addison-Wesley, Reading, MA, 1998)
[276] D. König, J.F. Böhme, Optimizing the CORDIC algorithm for processors with pipeline architectures, in *Signal Processing V: Theories and Applications*, ed. by L. Torres, E. Masgrau, M.A. Lagunas (Elsevier Science, Amsterdam, 1990)
[277] I. Koren, *Computer Arithmetic Algorithms* (Prentice-Hall, Englewood Cliffs, 1993)
[278] I. Koren, O. Zinaty, Evaluating elementary functions in a numerical coprocessor based on rational approximations. IEEE Trans. Comput. **39**(8), 1030–1037 (1990)
[279] P. Kornerup, C. Lauter, V. Lefèvre, N. Louvet, J.-M. Muller, Computing correctly rounded integer powers in floating-point arithmetic. ACM Trans. Math. Softw. **37**(1), 4:1–4:23 (2010)
[280] P. Kornerup, V. Lefevre, N. Louvet, J.-M. Muller, On the computation of correctly rounded sums. IEEE Trans. Comput. **61**(3), 289–298 (2012)
[281] P. Kornerup, D.W. Matula, Finite precision lexicographic continued fraction number systems, in *Proceedings of the 7th IEEE Symposium on Computer Arithmetic* (1985). Reprinted in [440]
[282] P. Kornerup, D.W. Matula, *Finite Precision Number Systems and Arithmetic* (Cambridge University Press, 2010). Cambridge Books Online
[283] K. Kota, J.R. Cavallaro, Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special-purpose processors. IEEE Trans. Comput. **42**(7), 769–779 (1993)
[284] W. Krämer, Inverse standard functions for real and complex point and interval arguments with dynamic accuracy. Comput. Suppl. **6**, 185–212 (1988)
[285] J. Kropa, Calculator algorithms. Math. Mag. **51**(2), 106–109 (1978)
[286] H. Kuki, W.J. Cody, A statistical study of the accuracy of floating-point number systems. Commun. ACM **16**(14), 223–230 (1973)
[287] U.W. Kulisch, Mathematical foundation of computer arithmetic. IEEE Trans. Comput. **C-26**(7), 610–621 (1977)
[288] U.W. Kulisch, W.L. Miranker, *Computer Arithmetic in Theory and Practice* (Academic Press, New York, 1981)
[289] O. Kupriianova, C. Lauter, A domain splitting algorithm for the mathematical functions code generator, in *48th Asilomar Conference on Signals, Systems and Computers*, Nov 2014, pp. 1271–1275
[290] O. Kupriianova, C. Lauter, Replacing branches by polynomials in vectorizable elementary functions, in *Book of abstracts for 16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics* (2014)
[291] T. Lang, E. Antelo, CORDIC-based computation of arccos and arcsin, in *ASAP'97, The IEEE International Conference on Application-Specific Systems, Architectures and Processors*. IEEE Computer Society Press, Los Alamitos, CA, July 1997
[292] T. Lang, E. Antelo, Cordic-based computation of arccos and $\sqrt{1-t^2}$. J. VLSI Signal Process. Syst. **25**(1), 19–38 (2000)
[293] T. Lang, E. Antelo, High-throughput CORDIC-based geometry operations for 3D computer graphics. IEEE Trans. Comput. **54**(3), 347–361 (2005)
[294] T. Lang, J.A. Lee, SVD by constant-factor-redundant CORDIC, in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 264–271
[295] T. Lang, J.-M. Muller, Bound on run of zeros and ones for algebraic functions, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, June 2001, pp. 13–20
[296] M. Langhammer, B. Pasca, Efficient floating-point polynomial evaluation on FPGAs, in *Field Programmable Logic and Applications (FPL'2013)* (2013)
[297] M. Langhammer and B. Pasca. Elementary function implementation with optimized sub range polynomial evaluation. In *Field Programmable Custom Computing Machines 2013 (FCCM'13)*, pages 202–205, 2013
[298] P.J. Laurent, *Approximation et Optimisation*. Enseignement des Sciences (in French). Hermann, Paris, France (1972)
[299] C. Lauter, M. Mezzarobba, Semi-automatic floating-point implementation of special functions, in *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, June 2015, pp. 58–65
[300] C.Q. Lauter, Basic building blocks for a triple-double intermediate format. Technical Report 2005-38, LIP, École Normale Supérieure de Lyon, Sept 2005
[301] C.Q. Lauter, *Arrondi Correct de Fonctions Mathématiques*. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France, Oct 2008. In French, http://www.ens-lyon.fr/LIP/web/

[302] C.Q. Lauter, V. Lefèvre, An efficient rounding boundary test for pow(x, y) in double precision. IEEE Trans. Comput. **58**(2), 197–207 (2009)
[303] D.-U. Lee, W. Luk, J. Villasenor, P. Cheung, Hierarchical segmentation schemes for function evaluation, in *Proceedings of the IEEE International Conference on Field-Programmable Technology*, Dec 2003, pp. 92–99
[304] D.-U. Lee, J.D. Villasenor, Optimized custom precision function evaluation for embedded processors. IEEE Trans. Comput. **58**(1), 46–59 (2009)
[305] V. Lefèvre, *Developments in Reliable Computing*, chapter An Algorithm that Computes a Lower Bound on the Distance Between a Segment and $\mathbb{Z}^2$ (Kluwer Academic Publishers, Dordrecht, 1999), pp. 203–212
[306] V. Lefèvre, *Moyens Arithmétiques Pour un Calcul Fiable*. Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France (2000)
[307] V. Lefèvre, New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, June 2005, pp. 68–75
[308] V. Lefèvre, J.-M. Muller, Worst cases for correct rounding of the elementary functions in double precision, in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, June 2001
[309] V. Lefèvre, J.-M. Muller, On-the-fly range reduction. J. VLSI Signal Process. **33**(1/2), 31–35 (2003)
[310] V. Lefèvre, J.-M. Muller, A. Tisserand, Towards correctly rounded transcendentals, in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic* (1997)
[311] V. Lefèvre, J.-M. Muller, A. Tisserand, Toward correctly rounded transcendentals. IEEE Trans. Comput. **47**(11), 1235–1243 (1998). Reprinted in [442]
[312] V. Lefèvre, D. Stehlé, P. Zimmermann, Worst cases for the exponential function in the IEEE 754r decimal64 format, in *Reliable Implementation of Real Number Algorithms: Theory and Practice*. Lecture Notes in Computer Sciences, vol. 5045 (Springer, Berlin, 2008), pp. 114–126
[313] A.K. Lenstra, H.W. Lenstra Jr., L. Lovász, Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982)
[314] R.-C. Li, Near optimality of Chebyshev interpolation for elementary function computation. IEEE Trans. Comput. **53**(6), 678–687 (2004)
[315] R.-C. Li, S. Boldo, M. Daumas, Theorems on efficient argument reduction, in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 129–136
[316] R.-C. Li, P. Markstein, J. Okada, J. Thomas, The libm library and floating-point arithmetic in hp-ux for itanium 2. Technical report, Hewlett-Packard Company (2002), http://h21007.www2.hp.com/dspp/files/unprotected/libm.pdf
[317] X. Li, J. Demmel, D.H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, D.J. Yoo, Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002)
[318] G. Lightbody, R. Woods, R. Walke, Design of a parameterizable silicon intellectual property core for QR-based RLS filtering. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **11**(4), 659–678 (2003)
[319] H. Lin, H.J. Sips, On-line CORDIC algorithms, in *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, Sept 1989, pp. 26–33
[320] H. Lin, H.J. Sips, On-line CORDIC algorithms. IEEE Trans. Comput. **39**(8) (1990)
[321] R.J. Linhardt, H.S. Miller, Digit-by-digit transcendental function computation. RCA Rev. **30**, 209–247 (1969). Reprinted in [439]
[322] G.L. Litvinov, Approximate construction of rational approximations and the effect of error autocorrection. Applications. Technical Report 8, Institute of Mathematics, University of Oslo, May 1993
[323] Z. Liu, K. Dickson, J. McCanny, Application-specific instruction set processor for SoC implementation of modern signal processing algorithms. IEEE Trans. Circuits Syst. I: Regul. Pap. **52**(4), 755–765 (2005)
[324] W. Luther, Highly accurate tables for elementary functions. BIT **35**, 352–360 (1995)
[325] T. Lynch, E.E. Swartzlander, A formalization for computer arithmetic, in *Computer Arithmetic and Enclosure Methods*, ed. by L. Atanassova, J. Hertzberger (Elsevier Science, Amsterdam, 1992), pp. 137–145
[326] A.J. MacLeod, Algorithm 757; miscfun, a software package to compute uncommon special functions. ACM Trans. Math. Softw. **22**(3), 288–301 (1996)
[327] N. Macon, A. Spitzbart, Inverses of vandermonde matrices. Am. Math. Mon. **65**(2), 95–100 (1958)
[328] K. Maharatna, S. Banerjee, E. Grass, M. Krstic, A. Troya, Modified virtually scaling-free adaptive CORDIC rotator algorithm and architecture. IEEE Trans. Circuits Syst. Video Technol. **15**(11), 1463–1474 (2005)
[329] K. Makino, M. Berz, Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **6**(3), 239–312 (2003)
[330] M.A. Malcolm, Algorithms to reveal properties of floating-point arithmetic. Commun. ACM **15**(11), 949–951 (1972)
[331] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, Hewlett-Packard professional books (Prentice-Hall, Englewood Cliffs, 2000)

[332] P. Markstein, Accelerating sine and cosine evaluation with compiler assistance, in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 137–140
[333] P.W. Markstein, Computation of elementary functions on the IBM RISC System/6000 processor. IBM J. Res. Dev. **34**(1), 111–119 (1990). Reprinted in [442]
[334] D.W. Matula, P. Kornerup, Finite precision rational arithmetic: Slash number systems. IEEE Trans. Comput. **34**(1), 3–18 (1985). Reprinted in [440]
[335] D.W. Matula, M.T. Panu, A prescale-lookup-postscale additive procedure for obtaining a single precision ulp accurate reciprocal, in *Proceedings of the 20th IEEE Symposium on Computer Arithmetic* (2011), pp. 177–183
[336] C. Mazenc, X. Merrheim, J.M. Muller, Computing functions cos^{-1} and sin^{-1} using CORDIC. IEEE Trans. Comput. **42**(1), 118–122 (1993)
[337] J.E. Meggitt, Pseudo division and pseudo multiplication processes. IBM J. Res. Dev. **6**, 210–226 (1962)
[338] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, K. Maharatna, 50 years of CORDIC: algorithms, architectures, and applications. IEEE Trans. Circuits Syst. I: Regul. Pap. **56**(9), 1893–1907 (2009)
[339] G. Melquiond, *De l'arithmétique d'intervalles à la certification de programmes (in French)*. Ph.D. thesis, École Normale Supérieure de Lyon, Nov 2006, http://www.ens-lyon.fr/LIP/Pub/PhD2006.php
[340] G. Melquiond, Proving bounds on real-valued functions with computations, in *Proceedings of the 4th International Joint Conference on Automated Reasoning*. Lecture Notes in Artificial Intelligence, vol. 5195, ed. by A. Armando, P. Baumgartner, G. Dowek (Sydney, Australia, 2008), pp. 2–17
[341] G. Melquiond, S. Pion, Formally certified floating-point filters for homogeneous geometric predicates. Theor. Inf. Appl. **41**(1), 57–69 (2007)
[342] X. Merrheim, *Bases discrètes et calcul des fonctions élémentaires par matériel (in French)*. Ph.D. thesis, École Normale Supérieure de Lyon and Université Lyon I, France, Feb 1994
[343] M. Mezzarobba, NumGfun: a package for numerical and analytic computation with D-finite functions, in *ISSAC '10*, ed. by S.M. Watt. ACM (2010), pp. 139146
[344] M. Mezzarobba, *Autour de l'évaluation numérique des fonctions D-finies (in French)*. Ph.d. dissertation, École Polytechnique, Palaiseau, France, Nov 2011
[345] P. Midy, Y. Yakovlev, Computing some elementary functions of a complex variable. Math. Comput. Simul. **33**, 33–49 (1991)
[346] O. Møller, Quasi double-precision in floating-point addition. BIT **5**, 37–50 (1965)
[347] P. Montgomery, Five, six, and seven-term Karatsuba-like formulae. IEEE Trans. Comput. **54**(3), 362–369 (2005)
[348] R.K. Montoye, E. Hokonek, S.L. Runyan, Design of the IBM RISC System/6000 floating-point execution unit. IBM J. Res. Dev. **34**(1), 59–70 (1990). Reprinted in [442]
[349] R.E. Moore, *Interval Analysis* (Prentice-Hall, Englewood Cliffs, 1963)
[350] R. Morris, Tapered floating point: a new floating-point representation. IEEE Trans. Comput. **20**(12), 1578–1579 (1971)
[351] C. Mouilleron, G. Revy, Automatic generation of fast and certified code for polynomial evaluation, in *Proceedings of the 20th IEEE Symposium on Computer Arithmetic* (2011), pp. 233–242
[352] J.-M. Muller, Discrete basis and computation of elementary functions. IEEE Trans. Comput. **C-34**(9) (1985)
[353] J.-M. Muller, *Méthodologies de calcul des fonctions élémentaires (in French)*. Ph.D. thesis, Institut National Polytechnique de Grenoble, France, Sept 1985
[354] J.-M. Muller, Une méthodologie du calcul hardware des fonctions élémentaires (in French). M2AN **20**(4), 667–695 (1986)
[355] J.-M. Muller, A few results on table-based methods. Reliable Comput. **5**(3), 279–288 (1999)
[356] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston (2010). ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9
[357] A. Munk-Nielsen, J.M. Muller, On-line algorithms for computing exponentials and logarithms, in *Proceedings of Europar'96, Lecture Notes in Computer Science 1124* (Springer, Berlin, 1996)
[358] S. Nakamura, Algorithms for iterative array multiplication. IEEE Trans. Comput. **C-35**(8) (1986)
[359] R. Nave, Implementation of transcendental functions on a numerics processor. Microprocessing Microprogramming **11**, 221–225 (1983)
[360] Y.V. Nesterenko, M. Waldschmidt, On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). Mat. Zapiski **2**, 23–42 (1996). Available in English at http://www.math.jussieu.fr/~miw/articles/ps/Nesterenko.ps
[361] I. Newton, Methodus fluxionum et serierum infinitarum, 1664–1671
[362] K.C. Ng, Argument reduction for huge arguments: good to the last bit. Technical report, SunPro (1992)
[363] K.C. Ng, K.H. Bierman, Getting the right answer for the trigonometric functions. SunProgrammer, Spring 1992
[364] S. Oberman, M.J. Flynn, Implementing division and other floating-point operations: a system perspective, in *Scientific Computing and Validated Numerics (Proceedings of SCAN'95)*, ed. by Alefeld, Frommer, and Lang (Akademie Verlag, Berlin, 1996), pp. 18–24

[365] S.F. Oberman, *Design issues in high performance floating point arithmetic units*. Ph.D. thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, Nov 1996
[366] S.F. Oberman, Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor, in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Apr 1999, pp. 106–115. Reprinted in [442]
[367] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955–1988 (2005)
[368] Y. Okabe, N. Takagi, S. Yajima, Log-depth circuits for elementary functions using residue number system. Electron. Commun. Jpn, Part 3, **74**, 8 (1991)
[369] F.W.J. Olver, P.R. Turner, Implementation of level-index arithmetic using partial table look-up, in *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, May 1987
[370] A.R. Omondi, *Computer Arithmetic Systems, Algorithms, Architecture and Implementations*. Prentice-Hall International Series in Computer Science (Englewood Cliffs, NJ, 1994)
[371] R.R. Osoroi, E. Antelo, J.D. Bruguera, J. Villalba, E. Zapata, Digit on-line large radix CORDIC rotator, in *Proceedings of the IEEE International Conference on Application Specific Array Processors (Strasbourg, France)*, ed. by P. Cappello, C. Mongenet, G.R. Perrin, P. Quinton, Y. Robert (IEEE Computer Society Press, Los Alamitos, CA, 1995), pp. 246–257
[372] A. Ostrowski, *On Two problems in Abstract Algebra Connected with Horner's Rule* (Academic Press, New York, 1954), pp. 40–48
[373] M.L. Overton, *Numerical Computing with IEEE Floating-Point Arithmetic* (SIAM, Philadelphia, 2001)
[374] V.Y. Pan, Methods of computing values of polynomials. Russ. Math Surv. **21**(1), 105–135 (1966)
[375] B. Parhami, Carry-free addition of recoded binary signed-digit numbers. IEEE Trans. Comput. **C-37**, 1470–1476 (1988)
[376] B. Parhami, Generalized signed-digit number systems: a unifying framework for redundant number representations. IEEE Trans. Comput. **39**(1), 89–98 (1990)
[377] B. Parhami, On the implementation of arithmetic support functions for generalized signed-digit number systems. IEEE Trans. Comput. **42**(3), 379–384 (1993)
[378] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs* (Oxford University Press, New York, 2000)
[379] M.S. Paterson, L.J. Stockmeyer, On the number of nonscalar multiplications necessary to evaluate polynomials. SIAM J. Comput. **2**(1), 60–66 (1973)
[380] G. Paul, M.W. Wilson, Should the elementary function library be incorporated into computer instruction sets? ACM Trans. Math. Softw. **2**(2) (1976)
[381] M. Payne, R. Hanek, Radian reduction for trigonometric functions. SIGNUM Newslett. **18**, 19–24 (1983)
[382] D. Phatak, T. Goff, I. Koren, Constant-time addition and simultaneous format conversion based on redundant binary representations. IEEE Trans. Comput. **50**(11), 1267–1278 (2001)
[383] D.S. Phatak, Comments on Duprat and Muller's branching CORDIC. IEEE Trans. Comput. **47**(9), 1037–1040 (1998)
[384] D.S. Phatak, Double step branching CORDIC: a new algorithm for fast sine and cosine generation. IEEE Trans. Comput. **47**(5), 587–602 (1998)
[385] G.M. Phillips, *Interpolation and Approximation by Polynomials*, CMS books in mathematics (Springer, New York, 2003)
[386] M. Pichat, Correction d'une somme en arithmétique à virgule flottante. Numer. Math. **19**, 400–406 (1972). In French
[387] S. Pion, *De la Géométrie Algorithmique au Calcul Géométrique*. Ph.D. thesis, Université de Nice Sophia-Antipolis, France, Nov 1999. In French
[388] M.J.D. Powell, *Approximation Theory and Methods* (Cambridge University Press, 1981)
[389] D.M. Priest, Algorithms for arbitrary precision floating point arithmetic, in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 132–144
[390] X. Qian, H. Zhang, J. Yang, H. Huang, J. Zhang, D. Fan, Circuit implementation of floating point range reduction for trigonometric functions, in *IEEE International Symposium on Circuits and Systems*, May 2007, pp. 3010–3013
[391] C.V. Ramamoorthy, J.R. Goodman, K.H. Kim, Some properties of iterative square-rooting methods using high-speed multiplication. IEEE Trans. Comput. **C-21**, 837–847 (1972). Reprinted in [439]
[392] E.M. Reingold, Establishing lower bounds on algorithms—a survey, in *Spring Joint Computer Conference* (1972), pp. 471–481
[393] G.W. Reitwiesner, Binary arithmetic. Adv. Comput. **1**, 231–308 (1960)
[394] E. Remez, Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation (in french). C.R. Académie des Sciences, Paris **198**, 2063–2065 (1934)
[395] N. Revol, F. Rouillier, Motivations for an arbitrary precision interval arithmetic and the MPFI library. Reliable Comput. **11**, 1–16 (2005)
[396] J.R. Rice, *The Approximation of Functions* (Addison-Wesley, Reading, 1964)

[397] T.J. Rivlin, *An Introduction to the Approximation of Functions* (Blaisdell Publishing Company, Walham, MA, 1969). Republished by Dover (1981)
[398] T.J. Rivlin, *Chebyshev Polynomials. From Approximation Theory to Algebra*. Pure and Applied Mathematics, 2nd edn. (Wiley, New York, 1990)
[399] J.E. Robertson, A new class of digital division methods. IRE Trans. Electron. Comput. **EC-7**, 218–222 (1958). Reprinted in [439]
[400] J.E. Robertson, The correspondence between methods of digital division and multiplier recoding procedures. IEEE Trans. Comput. **C-19**(8) (1970)
[401] D. Roegel, A reconstruction of the tables of Briggs' Arithmetica logarithmica (1624). Technical Report inria-00543939, Inria, France (2010), https://hal.inria.fr/inria-00543939
[402] S. Rump, F. Bunger, C.-P. Jeannerod, Improved error bounds for floating-point products and horners scheme. BIT Numer. Math. 1–15 (2015)
[403] S. Rump, F. Bunger, C.-P. Jeannerod, Improved error bounds for floating-point products and horners scheme. BIT Numer. Math. 1–15 (2015)
[404] S.M. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part II: sign, K-fold faithful and rounding to nearest. SIAM J. Sci. Comput. (2005–2008). Submitted for publication
[405] S.M. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part I: faithful rounding. SIAM J. Sci. Comput. **31**(1), 189–224 (2008)
[406] D.M. Russinoff, A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. LMS J. Comput. Math. **1**, 148–200 (1998)
[407] B.V. Sakar, E.V. Krishnamurthy, Economic pseudodivision processes for obtaining square root, logarithm and arctan. IEEE Trans. Comput. **C-20**(12) (1971)
[408] E. Salamin, Computation of π using arithmetic-geometric mean. Math. Comput. **30**, 565–570 (1976)
[409] D.D. Sarma, D.W. Matula, Faithful bipartite ROM reciprocal tables, in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, June 1995, pp. 17–28
[410] W.S. Sayed, H.A.H. Fahmy, What are the correct results for the special values of the operands of the power operation? ACM Trans. Math. Softw. (to appear)
[411] C.W. Schelin, Calculator function approximation. Am. Math. Mon. **90**(5) (1983)
[412] H. Schmid, A. Bogacki, Use decimal CORDIC for generation of many transcendental functions. *EDN*, pp. 64–73, Feb 1973
[413] A. Schönhage, V. Strassen, Schnelle Multiplikation Grosser Zahlen. Computing **7**, 281–292 (1971). In German
[414] M.J. Schulte, J. Stine, Symmetric bipartite tables for accurate function approximation, in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic* (1997)
[415] M.J. Schulte, J.E. Stine, Accurate function evaluation by symmetric table lookup and addition, in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (Zurich, Switzerland)* (1997), pp. 144–153
[416] M.J. Schulte, J.E. Stine, Approximating elementary functions with symmetric bipartite tables. IEEE Trans. Comput. **48**(8), 842–847 (1999)
[417] M.J. Schulte, E.E. Swartzlander, Exact rounding of certain elementary functions, in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, June 1993, pp. 138–145
[418] M.J. Schulte, E.E. Swartzlander, Hardware designs for exactly rounded elementary functions. IEEE Trans. Comput. **43**(8), 964–973 (1994). Reprinted in [442]
[419] P. Sebah, X. Gourdon, Newton's method and high-order iterations (2001), http://numbers.computation.free.fr/Constants/Algorithms/newton.html
[420] R.B. Seidensticker, Continued fractions for high-speed and high-accuracy computer arithmetic, in *Proceedings of the 6th IEEE Symposium on Computer Arithmetic* (1983), pp. 184–193
[421] A. Seznec, F. Lloansi, Étude des architectures des microproceseurs MIPS R10000, Ultrasparc et Pentium Pro (in French). Technical Report 1024, IRISA Rennes, France, May 1996
[422] A. Seznec, T. Vauléon, Étude comparative des architectures des microprocesseurs Intel Pentium et PowerPC 601 (in French). Technical Report 835, IRISA Rennes, France, June 1994
[423] J.R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. **18**, 305–363 (1997)
[424] R. Shukla, K.C. Ray, A low latency hybrid CORDIC algorithm. IEEE Trans. Comput. **63**(12), 3066–3078 (2014)
[425] J.D. Silverstein, S.E. Sommars, Y.C. Tao, The UNIX system math library, a status report, in *USENIX — Winter'90* (1990)
[426] A. Singh, D. Phatak, T. Goff, M. Riggs, J. Plusquellic, C. Patel, Comparison of branching CORDIC implementations, in *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, June 2003, pp. 215–225

[427] D. Smith, Efficient multiple-precision evaluation of the elementary functions. Math. Comput. **52**(185), 131–134 (1989)
[428] R.A. Smith, A continued-fraction analysis of trigonometric argument reduction. IEEE Trans. Comput. **44**(11), 1348–1351 (1995)
[429] W.H. Specker, A class of algorithms for $ln(x)$, $exp(x)$, $sin(x)$, $cos(x)$, $tan^{-1}(x)$ and $cot^{-1}(x)$. IEEE Trans. Electron. Comput. **EC-14** (1965). Reprinted in [439]
[430] H.M. Stark, *An Introduction to Number Theory* (MIT Press, Cambridge, 1981)
[431] D. Stehlé, *Algorithmique de la Réduction de Réseaux et Application à la Recherche de Pires Cas pour l'Arrondi de Fonctions Mathématiques (in French)*. Ph.D. thesis, Université Henri Poincaré—Nancy 1, France, Dec 2005
[432] D. Stehlé, V. Lefèvre, P. Zimmermann, Worst cases and lattice reduction, in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 142–147
[433] D. Stehlé, V. Lefèvre, P. Zimmermann, Searching worst cases of a one-variable function. IEEE Trans. Comput. **54**(3), 340–346 (2005)
[434] D. Stehlé, P. Zimmermann, Gal's accurate tables method revisited, in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, June 2005, pp. 257–264
[435] J.E. Stine, M.J. Schulte, The symmetric table addition method for accurate function approximation. J. VLSI Signal Process. **21**, 167–177 (1999)
[436] S. Story, P.T.P. Tang, New algorithms for improved transcendental functions on IA-64, in *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Apr 1999, pp. 4–11
[437] D.A. Sunderland, R.A. Strauch, S.W. Wharfield, H.T. Peterson, C.R. Cole, CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. IEEE J. Solid State Circuits **SC-19**(4), 497–506 (1984)
[438] T.Y. Sung, Y.H. Hu, Parallel VLSI implementation of Kalman filters. IEEE Trans. Aerosp. Electron. Syst, AES **23**(2) (1987)
[439] E.E. Swartzlander, *Computer Arithmetic*, vol. 1 (World Scientific Publishing Co., Singapore, 2015)
[440] E.E. Swartzlander, *Computer Arithmetic*, vol. 2 (World Scientific Publishing Co., Singapore, 2015)
[441] E.E. Swartzlander, A.G. Alexpoulos, The sign-logarithm number system. IEEE Trans. Comput. (1975). Reprinted in [439]
[442] E.E. Swartzlander, C.E. Lemonds, *Computer Arithmetic*, vol. 3 (World Scientific Publishing Co., Singapore, 2015)
[443] N. Takagi. *Studies on hardware algorithms for arithmetic operations with a redundant binary representation*. Ph.D. thesis, Dept. Info. Sci., Kyoto University, Japan (1987)
[444] N. Takagi, T. Asada, S. Yajima, A hardware algorithm for computing sine and cosine using redundant binary representation. Syst. Comput. Jpn. **18**(8) (1987)
[445] N. Takagi, T. Asada, S. Yajima, Redundant CORDIC methods with a constant scale factor. IEEE Trans. Comput. **40**(9), 989–995 (1991)
[446] N. Takagi, H. Yasukura, S. Yajima, High speed multiplication algorithm with a redundant binary addition tree. IEEE Trans. Comput. **C-34**(9) (1985)
[447] P.T.P. Tang, Table-driven implementation of the exponential function in IEEE floating-point arithmetic. ACM Trans. Math. Softw. **15**(2), 144–157 (1989)
[448] P.T.P. Tang, Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. ACM Trans. Math. Softw. **16**(4), 378–400 (1990)
[449] P.T.P. Tang, Table lookup algorithms for elementary functions and their error analysis, in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, June 1991, pp. 232–236
[450] P.T.P. Tang, Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. ACM Trans. Math. Softw. **18**(2), 211–222 (1992)
[451] The Polylib Team. Polylib, a library of polyhedral functions, version 5.20.0 (2004), http://icps.u-strasbg.fr/polylib/
[452] D.B. Thomas, A general-purpose method for faithfully rounded floating-point function approximation in FPGAs, in *Proceedings of the 22nd Symposium on Computer Arithmetic* (2015), pp. 42–49
[453] J. Thompson, N. Karra, M. Schulte, A 64-bit decimal floating-point adder, in *IEEE Computer society Annual Symposium on VLSI* (2004), pp. 297–298
[454] D. Timmermann, H. Hahn, B.J. Hosticka, Low latency time CORDIC algorithms. IEEE Trans. Comput. **41**(8), 1010–1015 (1992)
[455] D. Timmermann, H. Hahn, B.J. Hosticka, B. Rix, A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. INTEGRATION, VLSI J. **11**, 85–100 (1991)
[456] D. Timmermann, H. Hahn, B.J. Hosticka, G. Schmidt, A programmable CORDIC chip for digital signal processing applications. IEEE J. Solid-State Circuits **26**(9), 1317–1321 (1991)
[457] A.L. Toom, The complexity of a scheme of functional elements realizing the multiplication of integers. Sov. Math. Dokl. **3**, 714–716 (1963)
[458] L. Trefethen, *Approximation Theory and Approximation Practice* (Siam, 2013)
[459] L. Trefethen, Computing numerically with functions instead of numbers. Commun. ACM **58**(10), 91–97 (2015)

[460] C.-Y. Tseng, A multiple-exchange algorithm for complex chebyshev approximation by polynomials on the unit circle. SIAM J. Numer. Anal. **33**(5), 2017–2049 (1996)
[461] L. Veidinger, On the numerical determination of the best approximations in the Chebyshev sense. Numer. Math. **2**, 99–105 (1960)
[462] B. Verdonk, A. Cuyt, D. Verschaeren, A precision- and range-independent tool for testing floating-point arithmetic. I: basic operations, square root, and remainder. ACM Trans. Math. Softw. **27**(1), 92–118 (2001)
[463] B. Verdonk, A. Cuyt, D. Verschaeren, A precision- and range-independent tool for testing floating-point arithmetic. II: conversions. ACM Trans. Math. Softw. **27**(1), 119–140 (2001)
[464] J. Villalba, T. Lang, M. Gonzalez, Double-residue modular range reduction for floating-point hardware implementations. IEEE Trans. Comput. **55**(3), 254–267 (2006)
[465] J.E. Volder, The CORDIC computing technique. IRE Trans. Electron. Comput. **EC-8**(3), 330–334 (1959). Reprinted in [439]
[466] J.E. Volder, The birth of CORDIC. J. VLSI Signal Process. Syst. **25**(2), 101–105 (2000)
[467] J.E. Vuillemin, Exact real computer arithmetic with continued fractions. IEEE Trans. Comput. **39**(8) (1990)
[468] C.S. Wallace, A suggestion for a fast multiplier. IEEE Trans. Electron. Comput. 14–17 (1964). Reprinted in [439]
[469] P.J.L. Wallis (ed.), *Improving Floating-Point Programming* (John Wiley, New York, 1990)
[470] J.S. Walther, A unified algorithm for elementary functions, in *Joint Computer Conference Proceedings* (1971). Reprinted in [439]
[471] J.S. Walther, The story of unified CORDIC. J. VLSI Signal Process. Syst. **25**(2), 107–112 (2000)
[472] D. Wang, J.-M. Muller, N. Brisebarre, M. Ercegovac, (m, p, k)-friendly points: a table-based method to evaluate trigonometric function. IEEE Trans. Circuits Syst. II: Express Briefs **61**(9), 711–715 (2014)
[473] L. Wang, J. Needham, Horner's method in chinese mathematics: its origins in the root-extraction procedures of the han dynasty. T'oung Pao **43**(5), 345–401 (1955), http://www.jstor.org/stable/4527405
[474] S. Wang, E.E. Swartzlander, Merged CORDIC algorithm, in *1995 IEEE International Symposium on Circuits and Systems*, Apr 1995, pp. 1988–1991
[475] W.F. Wong, E. Goto, Fast hardware-based algorithms for elementary function computations using rectangular multipliers. IEEE Trans. Comput. **43**(3), 278–294 (1994)
[476] C.-S. Wu, A.-Y. Wu, C.-H. Lin, A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes. IEEE Trans. Circuits Syst. II: Analog Digit. Signal Process. **50**(9), 589–601 (2003)
[477] J.M. Yohe, Roundings in floating-point arithmetic. IEEE Trans. Comput. **C-22**(6), 577–586 (1973)
[478] H. Yoshimura, T. Nakanishi, H. Tamauchi, A 50MHz geometrical mapping processor, in *Proceedings of the 1988 IEEE International Solid-State Circuits Conference* (1988)
[479] T.J. Ypma, Historical development of the Newton-Raphson method. SIAM Rev. **37**(4), 531–551 (1995)
[480] P. Zimmermann, Arithmétique en précision arbitraire. Réseaux et Systèmes Répartis, Calculateurs Parallèlles **13**(4–5), 357–386 (2001). In French
[481] A. Ziv, Fast evaluation of elementary mathematical functions with correctly rounded last bit. ACM Trans. Math. Softw. **17**(3), 410–423 (1991)
[482] F. Zou, P. Kornerup, High speed DCT/IDCT using a pipelined CORDIC algorithm, in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 180–187
[483] D. Zuras, More on squaring and multiplying large integers. IEEE Trans. Comput. **43**(8), 899–908 (1994)

Index

J.-M. Muller, *Elementary Functions*, DOI 10.1007/978-1-4899-7983-4