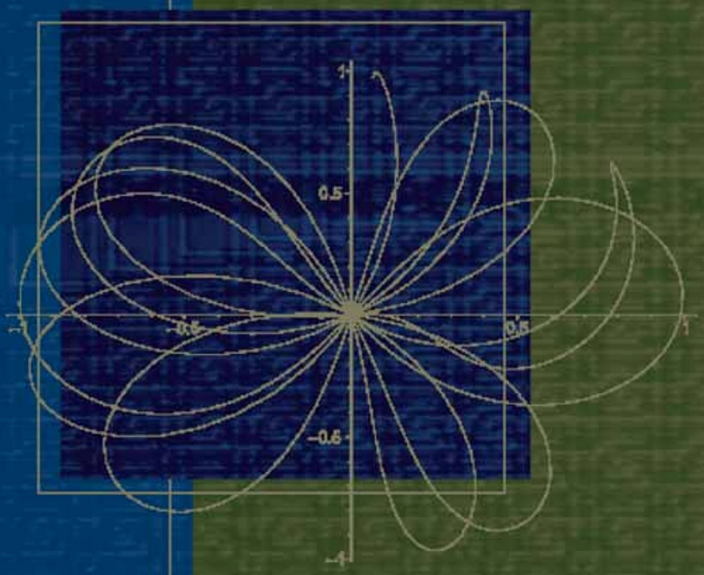
Elementary Functions

Algorithms and Implementation

SECOND EDITION



Jean-Michel Muller

BIRKHÄUSER

Jean-Michel Muller

Elementary Functions

Algorithms and Implementation

Second Edition

Birkhäuser
Boston • Basel • Berlin

Jean-Michel Muller
CNRS-Laboratoire LIP
Ecole Normale Superieure de Lyon
46 allee d'Italie
69364 Lyon Cedex 07
France

Cover design by Joseph Sherman.

AMS Subject Classifications: 26A09, 33Bxx, 90Cxx, 65D15, 65K05, 68Wxx, 65Yxx
ACM Subject Classifications: B.2.4, G.1.0., G.1.2, G.4

Library of Congress Cataloging-in-Publication Data
Muller, J. M. (Jean-Michel), 1961-
Elementary functions : algorithms and implementation / Jean-Michel Muller.– 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-8176-4372-9 (alk. paper)
1. Functions–Data processing 2. Algorithms. I. Title.
QA331.M866 2005
518′.1–dc22 2005048094

ISBN-10 0-8176-4372-9 eISBN 0-8176-4408-3 Printed on acid-free paper.
ISBN-13 978-0-8176-4372-0

Printed in the United States of America. (LAP/HP)

9 8 7 6 5 4 3 2 1

www.birkhauser.com

Contents

List of Figures

List of Tables

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 5 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.[1] 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 important when accuracy is a major goal (namely, range reduction, monotonicity and correct rounding). The third

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

one mainly deals with exceptions. The last chapter gives some examples of implementation.

A BIBTEX database containing the references in the bibliography, the Maple programs presented in the book and possible corrections are available over the Internet on the web page of the book:

www.springeronline.com/0-8176-4372-9.

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 and 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`).

Jean-Michel Muller
Lyon
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.'s 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 6), 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).

Jean-Michel Muller
Lyon
March 1997

Chapter 1

Introduction

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 [4] 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. [249], and Beame et al. [25]). 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 by Cody and Waite [64], and Hart et al. [160], 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. 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, and H. Kuki (see [2] for a review), have accustomed users to very accurate results (some existing implementations are graded in [29]). Some years ago a library providing elementary functions with one or two incorrect bits only was considered adequate [28], 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). Among the various properties that are desirable, one can cite:

- 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. [287], monotonicity failures can cause problems in evaluating divided differences;
- preservation of the direction of rounding: for instance, if the active rounding mode is round towards $-\infty$ (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 [287]

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

Let us deal with the problem of accuracy. The IEEE-754 standard for floating-point arithmetic (see Section 2.1) greatly helped to improve the reliability and portability of numerical software. And yet it says 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 [257]:

> *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 10 for an example). Also, correctly rounded functions are much

[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.

more difficult to implement. They 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

$$\overbrace{1.00000000010100111111100\cdots0011}^{53\text{ bits}}0\overbrace{111111111111\cdots111111111111}^{59\text{ ones}}010\cdots$$

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 is given in [103].

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 approximations of functions. Nowadays, software such as Maple[2] [57] readily computes such coefficients with very good accuracy, requiring a few minutes of CPU time on a PC or a workstation. 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, single-precision functions are programmed using double-precision arithmetic, or double-precision functions are programmed using double-extended precision 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.

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

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. However, that chapter cannot replace a textbook on computer arithmetic: someone who has to implement an elementary function on a circuit or an FPGA may have to choose between different addition/multiplication/division algorithms and architectures. Recent books devoted to computer arithmetic have been written by Swartzlander [298, 299], Koren [190], Omondi [250], Parhami [256], and Ercegovac and Lang [136]. Division and square-root algorithms and architectures are dealt with in a book by Ercegovac and Lang [135]. 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 theory of the approximation of functions by polynomials or rational functions goes back to the end of the 19th 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.

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*. Tabulating a function for all possible input values can be done for small word lengths (say, up to 16 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 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 [101]). This kind of method is efficient and widely used in current systems, but 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. The record holder at the time I am working on this edition is probably Y. Kanada. His team from Tokyo University computed the first $1,241,100,000,000$ decimal digits of π in 2002, using the following two formulas [12, 33]:

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

This required 600 hours of calculation on a parallel Hitachi computer with 64 processors.

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 17th 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 [324], 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 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.

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[3] with 4-digit mantissas, 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, 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] 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, and will be considered in the revision of the IEEE-754 Standard for floating-point arithmetic.

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-off 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.

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.

Chapter 2

Some Basic Things About Computer Arithmetic

2.1 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 Goldberg's paper [152], which gives a good survey of the topic, Kahan's lecture notes [182], which offer interesting and useful information, and Overton's [252] and Higham's [165] books. Further information can be found in [39, 66, 73, 74, 136, 157, 183, 190, 195, 250, 256, 320, 325]. Here we mainly focus on the IEEE-754 standard [5] for radix-2 floating-point arithmetic. The IEEE standard (and its follower, the IEEE-854 radix-independent standard [176]) 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] At the time I am preparing the second edition of this book, the IEEE Standard for floating-point arithmetic is under revision.[2]

2.1.1 Floating-point formats

Definition 1 *In a floating-point system of radix (or base) r, mantissa length n, and exponent range $E_{min} \ldots E_{max}$, a number t is represented by a mantissa, or significand $M_t = t_0.t_1t_2 \cdots t_{n-1}$ which is an n-digit number in radix r, satisfying $0 \leq M_t < r$, a sign $s_t = \pm 1$, and an exponent E_t, $E_{min} \leq E_t \leq E_{max}$, such that*

$$t = s_t \times M_t \times r^{E_t}.$$

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

[2] See `http://754r.ucbtest.org/`.

System	r	n	$E_{\min}$	$E_{\max}$	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	2	23+1	-126	127	$3.4\cdots \times 10^{38}$
	2	52+1	-1022	1023	$1.8\cdots \times 10^{308}$
IEEE-754-R "binary 128"	2	112+1	-16382	16383	$1.2\cdots \times 10^{4932}$
IEEE-754-R "decimal 64"	10	16	-383	384	$9.999\cdots 9 \times 10^{384}$

Table 2.1: *Basic parameters of various floating-point systems (n is the size of the mantissa, expressed in number of digits in the radix of the computer system). The "+1" is due to the hidden bit convention. The values concerning IEEE-754-R may change: the standard is under revision. The binary 32 and binary 64 formats of IEEE-754-R are the same as the single- and double-precision formats of IEEE-754.*

For accuracy reasons it is frequently required that the floating-point representations be *normalized*, that is, that the mantissas be greater than or equal to 1. This requires a special representation for the number zero.

An interesting consequence of that normalization, for radix 2, is that the first mantissa (or significand) digit of a floating-point nonzero number must always be "1." Therefore there is no need to store it, 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 [165, 182, 190, 250]. For instance, the largest representable finite number in the IEEE-754 double-precision format [5] is

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

and the smallest positive normalized number is

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

Arithmetic based on radix 10 is used in pocket calculators[3]. Also, it is still used in financial calculations, and it remains an object of study [76]. A Russian

[3]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

computer named SETUN [51] used radix 3 with digits -1, 0 and 1 (this is called the *balanced ternary system*). It was built[4] at Moscow University, during the 1960s [187]. Almost all other current computing systems use base 2. Various studies [39, 66, 195] 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 — cannot benefit from being done in a higher radix *inside* the arithmetic operators).

2.1.2 Rounding modes

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 an *active rounding mode* from:

- rounding towards $-\infty$: $\nabla(x)$ is the largest machine number less than or equal to x;
- rounding towards $+\infty$: $\Delta(x)$ is the smallest machine number greater than or equal to x;
- rounding towards 0: $\mathcal{Z}(x)$ is equal to $\nabla(x)$ if $x \geq 0$, and to $\Delta(x)$ if $x < 0$;
- rounding to the nearest: $\mathcal{N}(x)$ is the machine number that is the closest to x (with a special convention if x is exactly between two machine numbers: the chosen number is the "even" one, i.e., the one whose last mantissa bit is a zero).

This is illustrated using the example in Figure 2.1.

If the active rounding mode is denoted by $\diamond$, and u and v are machine numbers, then the IEEE-754 standard [5, 80] 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 "exactly rounded." There is a similar requirement for

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.

[4] See `http://www.icfcst.kiev.ua/MUSEUM/PHOTOS/setun-1.html`.

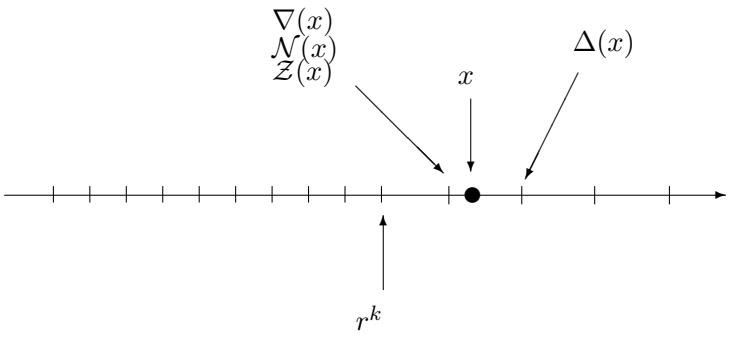


Figure 2.1: *Different possible roundings of a real number x in a radix-r floating-point system. In this example, $x > 0$.*

the square root. Such a requirement has a number of advantages:

- it leads to *full compatibility*[5] 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 arbitrary precision arithmetic [263], implementing "distillation" (i.e., obtaining the exact sum of several floating-point numbers) [181, 262, 263], or making decisions in computational geometry;
- one can easily implement *interval arithmetic* [196, 197, 233], or more generally one can get lower or upper bounds on the exact result of a sequence of arithmetic operations.

A very useful result that can be proved assuming correct rounding is the following algorithm (the first ideas that underlie it go back to Møller [231]).

Theorem 1 (Fast2Sum algorithm) *(Theorem C of [187], page 236). Assume the radix r 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. Here $\mathcal{N}(x)$ means x rounded 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. The following algorithm computes two floating-point numbers s and t that satisfy:*

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

[5] At least in theory: some compilers have a regrettable tendency to change the order of execution of operations for the sake of optimization. For more information on that problem, see Kahan's lecture notes [182].

Algorithm 1 (Fast2Sum(a,b))

$$\begin{aligned} s &:= \mathcal{N}(a+b); \\ z &:= \mathcal{N}(s-a); \\ t &:= \mathcal{N}(b-z). \end{aligned}$$

Unfortunately, there is no correct rounding requirement for the elementary functions, probably because it has been believed for many years that correct rounding of the elementary functions would be much too expensive. We analyze this problem in Chapter 10. 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.

2.1.3 Subnormal numbers and exceptions

In the IEEE-754 floating-point standard, numbers are normalized unless they are very small. *Subnormal* numbers (also called *denormalized numbers*) are nonzero numbers with a non-normalized mantissa and the smallest possible exponent (i.e., the exponent used for representing zero). This allows underflow to be gradual (see Figure 2.2). The minimum subnormal positive number in the IEEE-754 double-precision 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 2 (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 $x - y$ will be computed exactly.

That result is 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 $\sqrt{-5}$, ∞/∞, $+\infty + (-\infty)$, ...;
- $\pm\infty$ can be the result of an overflow, or the exact result of a division by zero; and

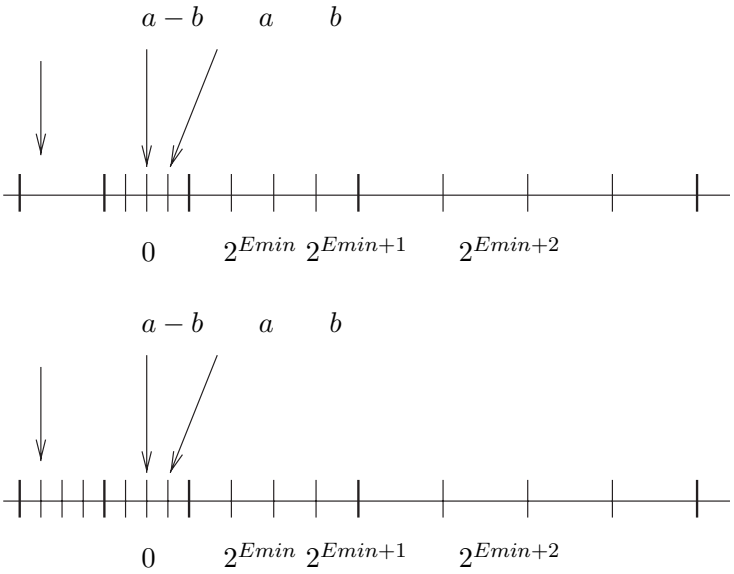


Figure 2.2: *Above is the set of the nonnegative, normalized floating-point numbers (assuming radix 2 and 2-bit mantissas). In that set, $a - b$ is not exactly representable, and the floating-point computation of $a - b$ will return 0 in the round to nearest, round to 0 or round to $-\infty$ rounding modes. Below, 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.*

- ± 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 [182] 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 occur as input values, and the circuit/library must be able to produce them as output values when needed.

2.1.4 ULPs

If x is *exactly representable* in a floating-point format and is not an integer power of the radix r, the term $\text{ulp}\,(x)$ (for *unit in the last place*) denotes the magnitude of the last mantissa digit of x. That is, if,

$$x = \pm x_0.x_1x_2\cdots x_{n-1} \times r^{E_x}$$

then $\text{ulp}\,(x) = r^{E_x-n+1}$. 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. And yet, as noticed by

Harrison [157], there are several incompatible definitions in the literature (they differ near the powers of r), and they sometimes have counterintuitive properties. In this edition, I will follow Harrison's definition, which slightly differs from the definition I gave in the first edition of this book. The only modification to Harrison's definition is the handling of numbers larger than the largest finite floating-point number. See [237] for a discussion of various definitions of function $\mathrm{ulp}(x)$ and their properties.

Definition 2 (Unit in the last place [157, 237]) *If x lies between two finite consecutive floating-point numbers a and b without being equal to one of them, then $\mathrm{ulp}(x) = |b - a|$, otherwise $\mathrm{ulp}(x)$ is the distance between the two finite floating-point numbers nearest x.*

The major advantage of this definition 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. This definition assumes that x is a real number. If x is a floating-point number we must add the requirement $\mathrm{ulp}(\mathrm{NaN}) = \mathrm{NaN}$, as suggested by Kahan.

2.1.5 Fused multiply-add operations

Some processors (e.g., the IBM PowerPC or the Intel/HP Itanium [83]) have a *fused multiply-add* (FMA, or *fused multiply-accumulate*, or *fused MAC*) instruction, that allows to compute $ax \pm b$, where a, x and b are floating-point numbers, with one final rounding only. Such an instruction may be extremely helpful for the designer of arithmetic algorithms:

- it facilitates the exact computation of remainders, which allows the design of efficient software for correctly rounded division [48, 83, 84, 182, 224, 226];
- it makes the evaluation of polynomials faster and – in general – more accurate: when using Horner's scheme[6], 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 interestings book to the evaluation of elementary functions using the fused multiply-add operations that are available on the HP/Intel Itanium processor [83, 224];
- as noticed by Karp and Markstein [185], it makes it possible to easily get the exact product of two floating-point variables. If $\mathcal{N}(x)$ is x rounded to the nearest, then from two floating-point numbers a and b, the following

[6] 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.

algorithm returns two values p and ρ such that p is the floating-point number that is closest to ab, and $p+\rho = ab$ exactly. It requires one multiplication and one fused multiply-add. Although I present it with round-to-nearest mode, it works as well with the other rounding modes.

Algorithm 2 (Fast2Mult(a,b))

$$p := \mathcal{N}(ab);$$
$$\rho := \mathcal{N}(ab - p)$$

Performing a similar calculation without a fused multiply-add operation is possible [105] but requires many more floating-point operations. Some other interesting arithmetic functions are easily implementable when a fused multiply-add is available [31, 47].

And yet, as noticed by Kahan [182] 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 a NaN when $x = y$. Consider as an example

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

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

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

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

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

Hence, if the floating-point computation of $x^2 - y^2$ is implemented as $((x^2) - y \times y)$, the obtained result will be less than 0 and computing its square root will generate a NaN, whereas the exact result is 0.

2.1.6 Testing your computational environment

The various parameters (radix, mantissa and exponent lengths, rounding modes,...) 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 [151, 223], that returns the radix of the floating-point system being used.

```
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 inquiry programs such as MACHAR [69] and PARANOIA [186], 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`), a recent tool presented by Verdonk, Cuyt and Verschaeren [315, 316], and the MPCHECK program written by Revol, Pélissier and Zimmermann (available at `http://www.loria.fr/~zimmerma/free/`).

2.1.7 Floating-point arithmetic and proofs

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 [157] and check floating-point trigonometric functions [158] for the Intel-HP IA64 architecture. Russinoff [272] 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 [30, 213].

2.1.8 Maple programs that compute double-precision approximations

The following Maple program implements the round-to-nearest-even rounding mode in double-precision. It computes the double-precision floating-point number that is closest to t for any real number t.

```
nearestdouble := proc(t)
local x, sign, logabsx, exponent, mantissa, infmantissa;
Digits := 100;
x := evalf(t);
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)
         expmin := -1022; expmax := 1024;
         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
# 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 program evaluates $\mathrm{ulp}\,(t)$ for any real number t.

```
ulp := proc(t)
local x;
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
       if x = 2^expmin then res := 2^(expmin-53)
        else res := 2^(expmin-52)
       fi;
    fi;
fi;
res;
end;
```

2.2 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 [52]. Booth recoding [32] (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 [135, 270] 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 $s = s_n s_{n-1} s_{n-2} \cdots s_0$ of two integers $x = x_{n-1} x_{n-2} \cdots x_0$ and $y = y_{n-1} y_{n-2} \cdots y_0$ 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"):

$$\begin{aligned} c_0 &= 0 \\ s_i &= x_i \oplus y_i \oplus c_i \\ c_{i+1} &= x_i y_i \vee x_i c_i \vee y_i c_i \end{aligned} \tag{2.1}$$

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 [136, 144, 190, 250, 256] and implemented in current microprocessors are much faster than a straightforward, purely sequential, implementation of (2.1). Nevertheless, the dependency relation between the carries makes a fully parallel addition impossible in the conventional number systems.

2.2.1 Signed-digit number systems

In 1961, Avizienis [11] 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 $+5$, every number is representable. The number 15725 can be represented by the digit chain $2\overline{4}3\overline{2}5$ (we use $\overline{4}$ to represent the digit -4); that is, $15725 = 2 \times 10^4 + (-4) \times 10^3 + (-3) \times 10^2 + 2 \times 10^1 + 5$.

The same number can also be represented by the digit chain $2\overline{4}3\overline{3}\overline{5}$. 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. The following algorithm 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[7] $2a \geq r + 1$.

Algorithm 3 (Avizienis)
Input : $x = x_{n-1}x_{n-2}\cdots x_0$ *and* $y = y_{n-1}y_{n-2}\cdots y_0$
Output : $s = s_n s_{n-1} s_{n-2}\cdots s_0$

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

$$\begin{cases} t_{i+1} = \begin{cases} +1 & \text{if } x_i + y_i \geq a \\ 0 & \text{if } -a+1 \leq x_i + y_i \leq a-1 \\ -1 & \text{if } x_i + y_i \leq -a \end{cases} \\ w_i \quad = x_i + y_i - r \times t_{i+1}. \end{cases} \tag{2.2}$$

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, on-line arithmetic [137], 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 non-redundant number system, but the internal

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

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

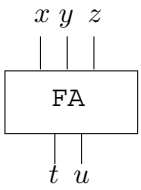
Figure 2.3: *Computation of* $1\overline{5}31\overline{2}0 + 1\overline{1}\overline{2}61\overline{6}$ *using Avizienis' algorithm in radix* $r = 10$ *with* $a = 6$.

calculations are performed in a redundant number system). For instance, most multipliers use (at least implicitly) the carry-save number system, whereas the divider of the Pentium actually uses two different redundant number systems: the division iterations are performed in carry-save, and the quotient is first generated in radix 4 with digits between -2 and $+2$, then converted to the usual radix-2 number system. This is typical when implementing an SRT division algorithm [135]. The reader interested in redundant number systems can find useful information in [11, 136, 253, 254, 255, 259].

2.2.2 Radix-2 redundant 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 [253]: 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 [61, 300]). Those two number systems allow very fast additions and subtractions. The *carry-save adder* (see, for instance, [190]) 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.

An adder structure for the borrow-save number system can easily be built using elementary cells slightly different from the FA cell. Let us present the algorithm for adding two BS numbers.



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

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

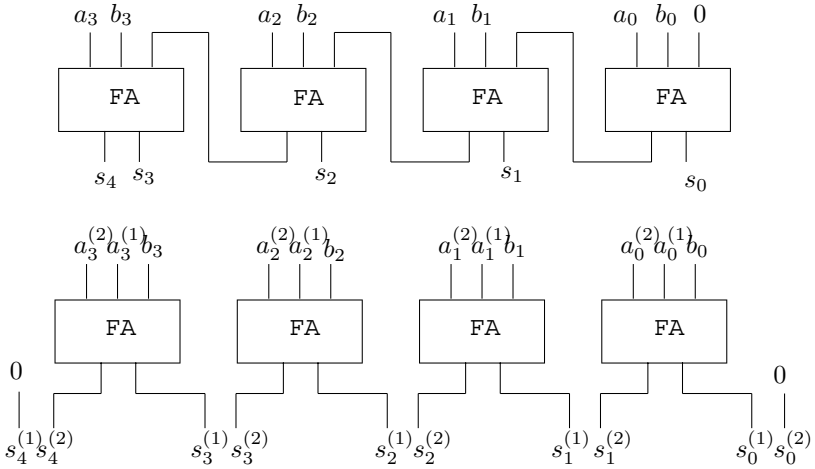


Figure 2.5: *A carry-save adder (bottom), compared to a carry-propagate adder (top).*

Algorithm 4 (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

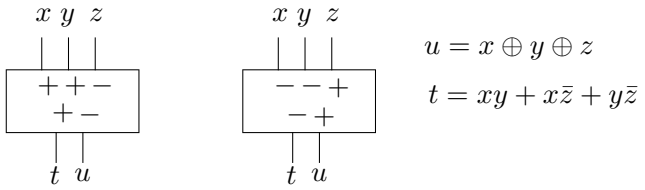


Figure 2.6: *A PPM cell. From three bits x, y, and z, it computes two bits t and u such that $x + y - z = 2t - u$.*

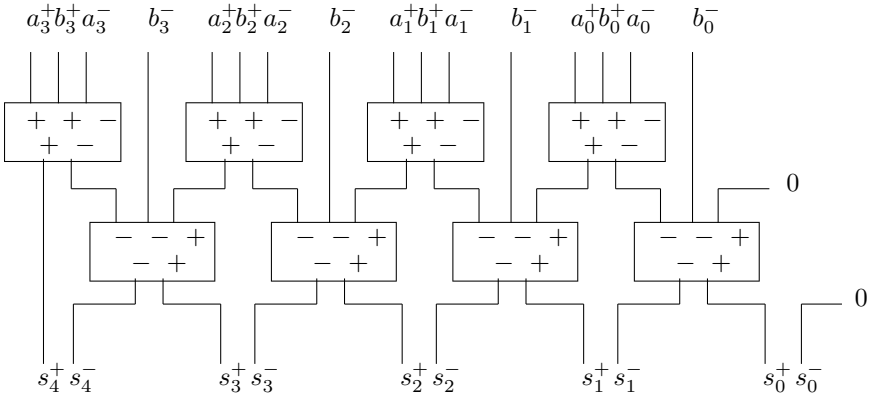


Figure 2.7: *A borrow-save adder.*

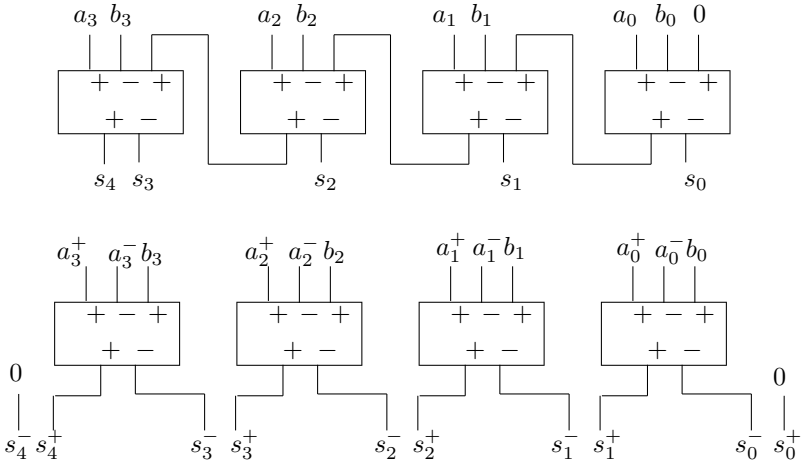


Figure 2.8: *A structure for adding a borrow-save number and a nonredundant number (bottom), compared to a carry-propagate subtractor (top).*

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, non-redundant, binary system by using only one row of PPM cells.[8] This is described in Figure 2.8. More details on borrow-save based arithmetic operators can be found in [17].

[8]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.

Chapter 3

Polynomial or Rational Approximations

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 function are evaluated in finite precision floating-point arithmetic. Can we compute tight bounds on these errors?

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

$$||p^* - f||_2 = \sqrt{\int_a^b w(x) \left(f(x) - p^*(x)\right)^2 dx},$$

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

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

3.1 Least Squares Polynomial Approximations

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

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

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

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

The approximation p^* can be computed as follows:

- Build a sequence (T_m), $(m \le n)$ of polynomials such that (T_m) is of degree m, and such that $\langle T_i, T_j \rangle = 0$ for $i \ne j$. 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 [150]. 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, 149].

[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.

3.1.1 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:

$$\langle T_i, T_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ \dfrac{2}{2i+1} & \text{otherwise.} \end{cases}$$

3.1.2 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:

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

An example of a Chebyshev polynomial (T_7) is plotted in Fig. 3.1.

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 [36, 269].

Theorem 3 *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.

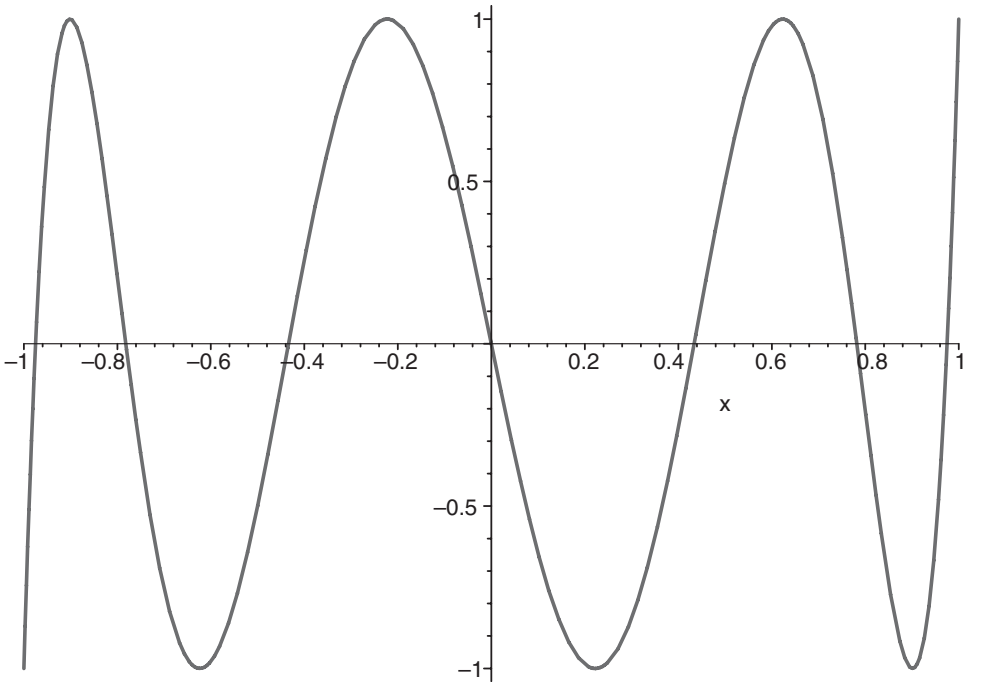


Figure 3.1: *Graph of the polynomial* $T_7(x)$.

Theorem 4 *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 5 (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

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

3.1.3 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

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

3.1.4 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:

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

3.1.5 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

$$p^*(x) = q^*\left(\frac{2}{b-a}x - \frac{a+b}{b-a}\right).$$

3.2 Least Maximum Polynomial Approximations

As in the previous section, we want to approximate a 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, $||f - p||_\infty$ denotes the *distance*:

$$||f - p||_\infty = \max_{a \le x \le b} |f(x) - p(x)|.$$

We look for a polynomial p^* that satisfies:

$$||f - p^*||_\infty = \min_{p \in \mathcal{P}_n} ||f - p||_\infty.$$

The polynomial p^* 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 6 (Weierstrass, 1885) *Let f be a continuous function. For any $\epsilon > 0$ there exists a polynomial p such that $||p - f||_\infty \le \epsilon$.*

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

Theorem 7 (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:

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

This theorem is illustrated in Figures 3.2 and 3.3 for the case $n = 3$.

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 property allows us to directly find p^* in some particular cases, as we show in Section 3.3. It is used by an algorithm, due to Remez [160, 265] (see Section 3.5), that computes the minimax degree-n approximation to a continuous function iteratively. The reader can consult the seminal work by de La Vallée Poussin [99], Rice's book [267], and a survey by Fraser [146].

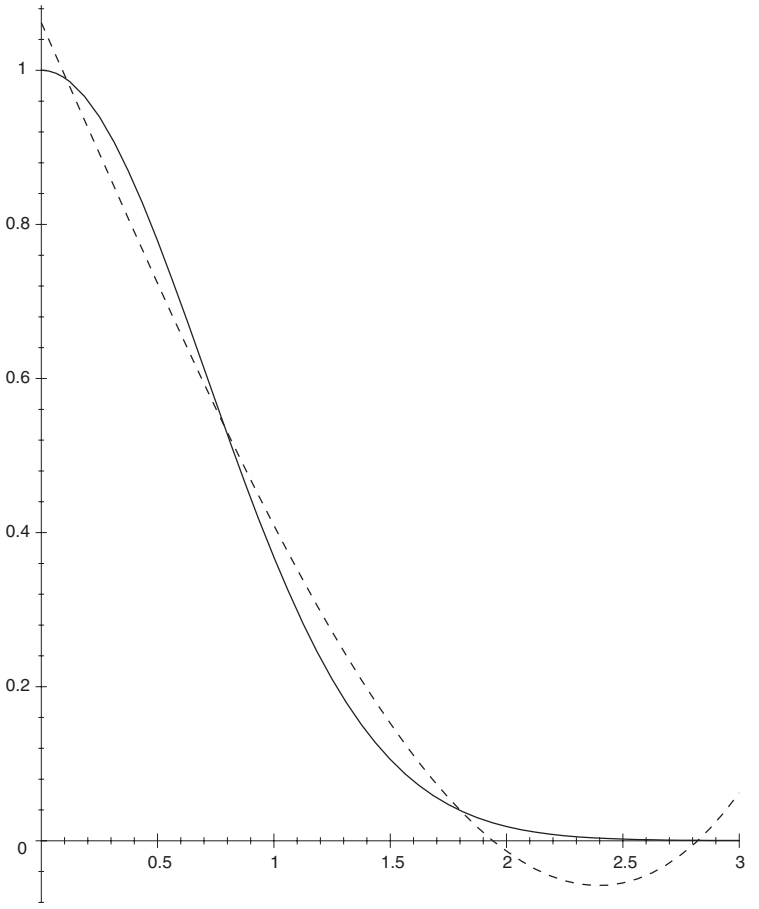


Figure 3.2: *The* $\exp(-x^2)$ *function and its degree-*3 *minimax approximation on the interval* $[0, 3]$ *(dashed line). There are five values where the maximum approximation error is reached with alternate signs.*

It is worth noticing that in some cases, $p^* - f$ may have more than $n + 2$ extrema. Figure 3.4 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 $[0, 4\pi]$.

3.3 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.*

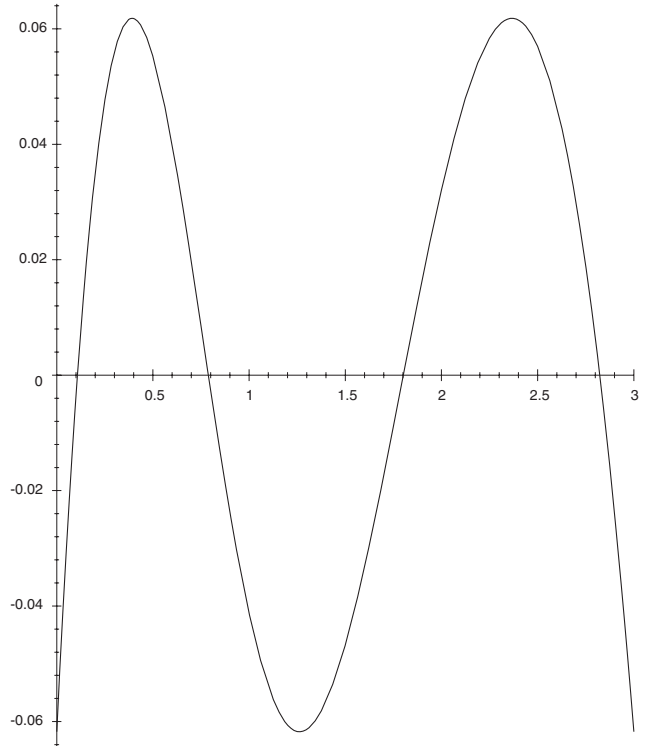


Figure 3.3: *The difference between the* $\exp(-x^2)$ *function and its degree-*3 *minimax approximation on the interval* $[0, 3]$.

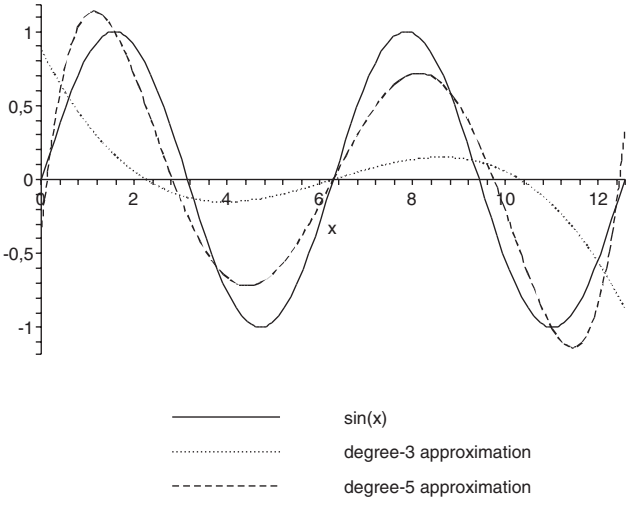


Figure 3.4: *The minimax polynomial approximations of degrees* 3 *and* 5 *to* $\sin(x)$ *in* $[0, 4\pi]$. *Notice that* $\sin(x) - p_3(x)$ *has* 6 *extrema. From Chebyshev's theorem, we know that it must have at least* 5 *extrema.*

Least squares approximation using Legendre polynomials

The first three Legendre polynomials are:

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

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 $p^*(x) = a_0 + a_1 x + a_2 x^2$ is the minimax approximation to e^x on $[-1, 1]$. From Theorem 7, 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:

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

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 be used for global (i.e., on an interval) approximations. The differences between the exponential function and its approximants are plotted in Figure 3.5: 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.

As shown in the previous example, Taylor expansions generally give poor polynomial approximations when the degree is sufficiently high, and should be avoided.[2] Let us consider another example. We wish to approximate the

[2] An exception is multiple-precision computations (see Chapter 5), since it is not possible to pre-compute and store least-squares or minimax approximations for all possible precisions.

	Taylor	Legendre	Chebyshev	Minimax
Max. Error	0.218	0.081	0.050	0.045

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

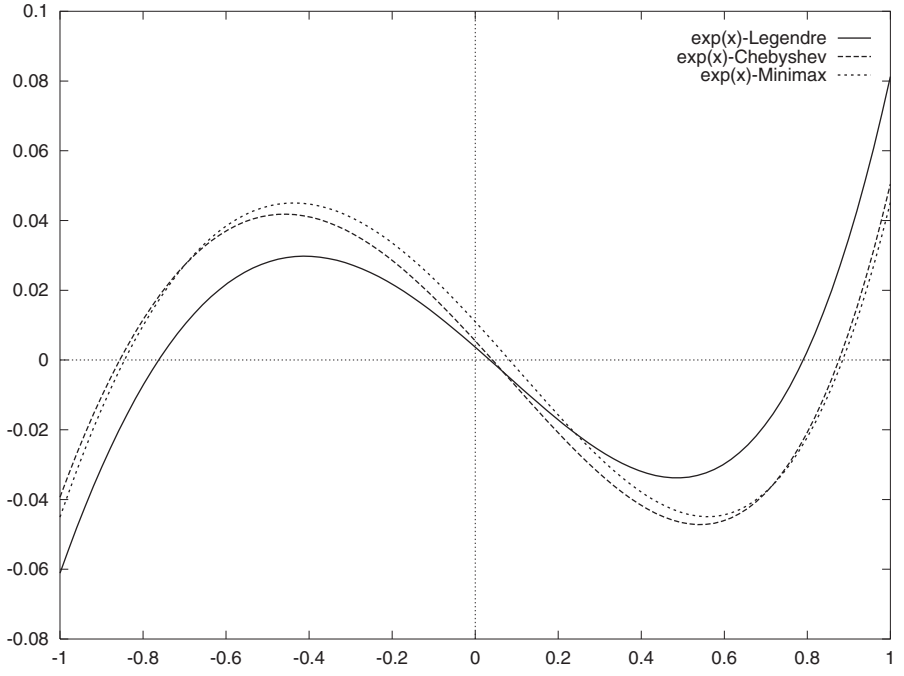


Figure 3.5: *Errors of various degree-2 approximations to* e^x *on* $[-1, 1]$. *Legendre approximation is better on average, and Chebyshev approximation is close to the minimax approximation.*

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. [160], 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 3.5). Roughly

speaking, if f is regular enough, the coefficients

$$a_i = \frac{\langle f, T_i \rangle}{\langle T_i, T_i \rangle} = \frac{\displaystyle\int_{-1}^{+1} \frac{f(x)T_i(x)}{\sqrt{1-x^2}} dx}{\displaystyle\int_{-1}^{+1} \frac{(T_i(x))^2}{\sqrt{1-x^2}} dx}$$

quickly decrease, so $a_{n+1}T_{n+1}$ is close to the difference between f and its degree-n Chebyshev approximation p_n^*, that is, $\sum_{i=0}^{n} a_i T_i$. Since $T_{n+1}(x) = \cos\left((n+1)\cos^{-1} x\right)$, the maximum value of T_{n+1} between -1 and 1 is reached $n+2$ times, with alternate signs. Therefore p_n^* "almost" satisfies the condition of Theorem 7. Of course, this is a rough explanation, not a proof. A proof can be given in some cases: Li [212] 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 a "general" and irregular enough function, the Chebyshev approximation can be rather far from the minimax approximation: this is illustrated by the next example.

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 3.6, and the worst-case errors are presented in Table 3.2.

Table 3.2 and Figure 3.6 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).

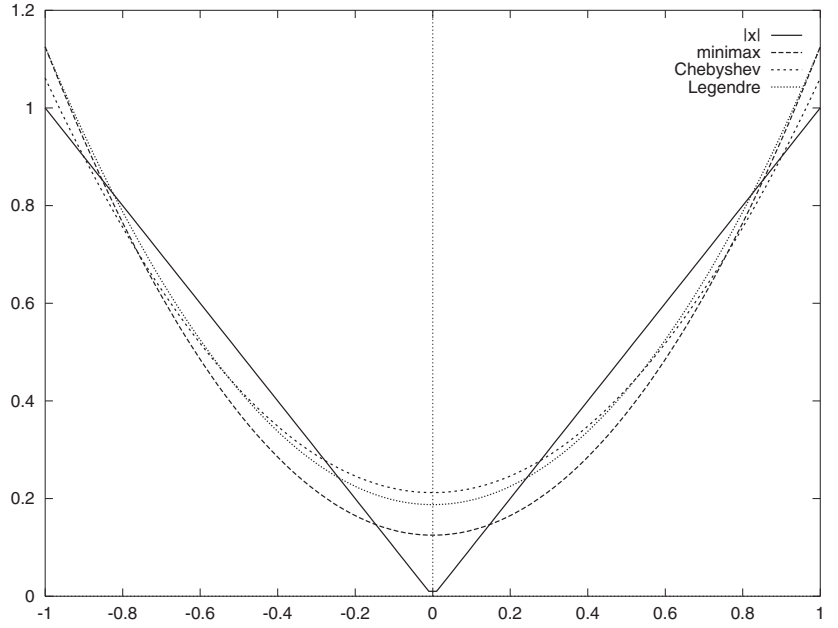


Figure 3.6: *Comparison of Legendre, Chebyshev, and minimax degree-2 approximations to* $|x|$.

	Legendre	Chebyshev	Minimax
Max. Error	0.1875	0.2122	0.125

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

3.4 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, to reach a given approximation error, the degree of the required approximation polynomial may be quite large. A theorem due to Bernstein [160] shows that the convergence of the degree-n minimax approximations towards 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 is difficult to predict. Figure 3.7 plots the figures given in the table.

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

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.*

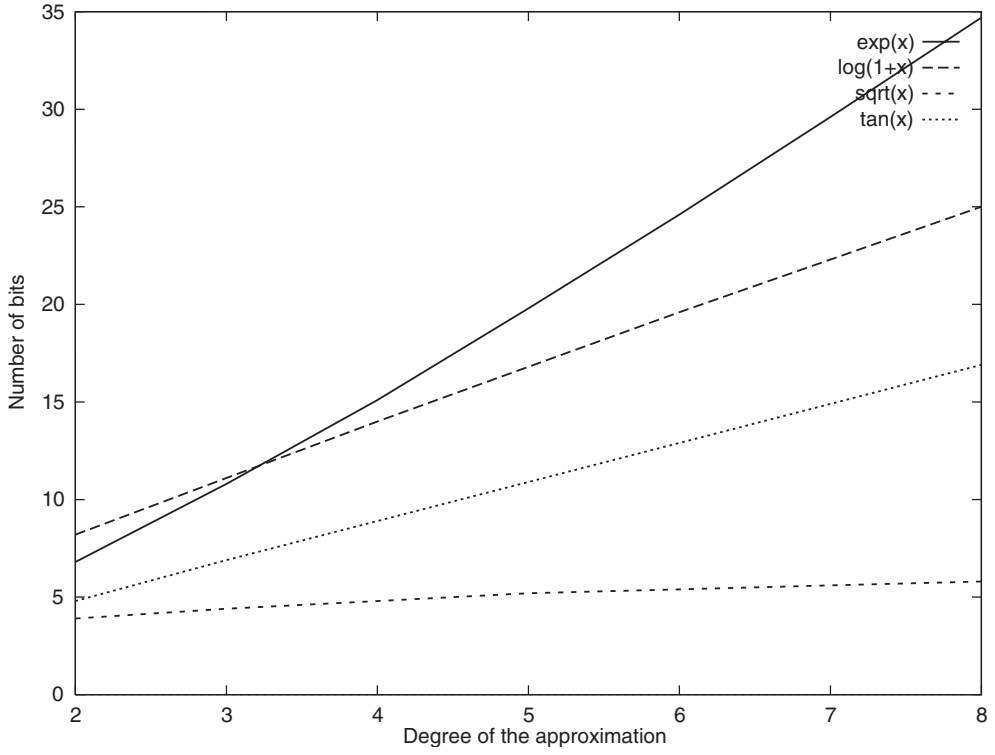


Figure 3.7: *Number of significant bits (obtained as* $-\log_2(error)$*) of the minimax polynomial approximations to various functions on* $[0,1]$*.*

3.5 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 [146]. 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 packages such as Maple (see Section 3.7) 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, ..., x_{n+1} of Theorem 7. We proceed as follows.

1. We start from an initial set of points x_0, x_1, ..., x_{n+1} in $[a, b]$.

2. We consider the linear system of equations

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

   It is a system of $n + 2$ linear equations, with $n + 2$ unknowns: p_0, p_1, ..., p_n and ϵ. Therefore in all non-degenerated cases, it will have 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 [146] that this is a convergent process, and that the speed of convergence is quadratic [314]. In general, starting from the initial set of points

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

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 sub-intervals
# and assumes each sub-interval 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 $|T_5(x-1)| = 1$.

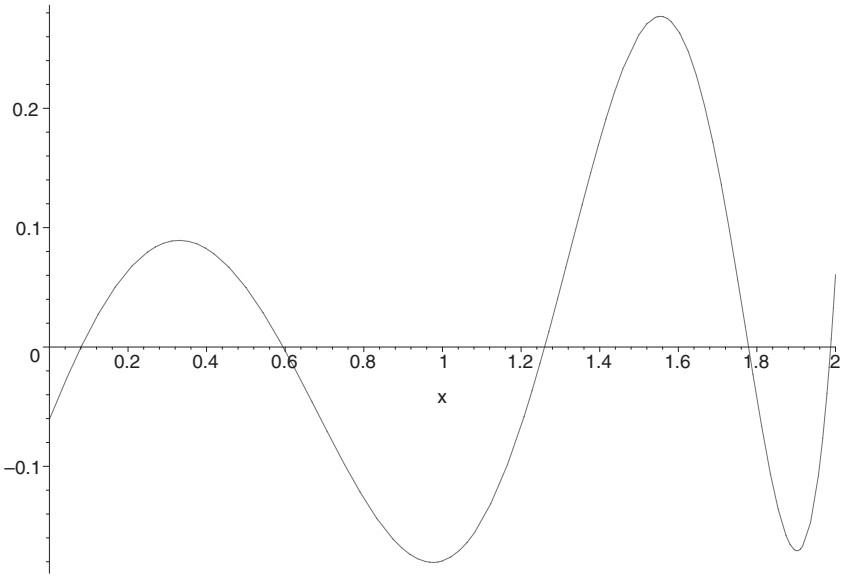


Figure 3.8: *Difference between* $P^{(1)}(x)$ *and* $\sin(\exp(x))$ *on* $[0, 2]$.

The corresponding linear system is

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

Solving this system gives the following polynomial:

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

The difference $P^{(1)}(x) - \sin(\exp(x))$ is plotted in Figure 3.8.

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,

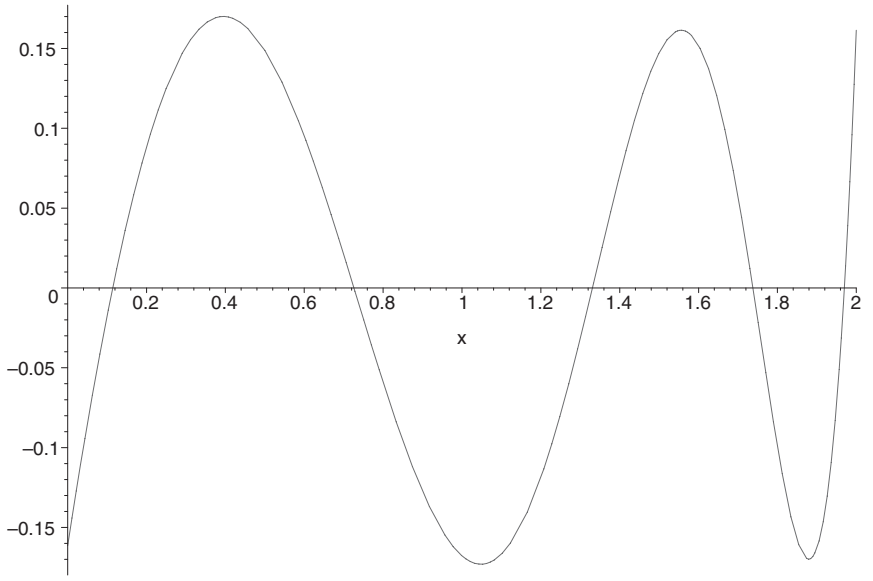


Figure 3.9: *Difference between* $P^{(2)}(x)$ *and* $\sin(\exp(x))$ *on* $[0, 2]$.

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 3.9. 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 Theorem 7. 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

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

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

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

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.

3.6 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[3] 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 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 domains, polynomial approximations may not

[3]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 [135] are preferable.

be suitable. One has to try *rational approximations*.[4] Concerning rational approximations, there is a characterization theorem, similar to Theorem 7, 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 8 (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*

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

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 [160, 219] for more details. Litvinov [219] 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* [19, 20], but such approximants have the same drawbacks as Taylor expansions: they are *local* (i.e., around one value) approximations only[5]. Algorithms that give "nearly best" approximations (even in regions of the complex plane) are given in [125]. There also exists a notion of *orthogonal rational functions* [50, 113]. See [27] 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.

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

$$\begin{aligned} &1.00000014609x + 0.333324808x^3 + 0.13347672x^5 + 0.0529139x^7 \\ &+0.0257829x^9 + 0.0013562x^{11} + 0.010269x^{13} \end{aligned}$$

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

[5] And yet, they can have better global behaviour than expected. See for instance reference [143].

Processor	FP Add	FP Mult	FP Div
Pentium	3	3	39
Pentium Pro	3	5	18-38
Pentium III	3	5	32
Pentium IV	5	7	38
PowerPC 601	4	4	31
PowerPC 750	3	4	31
MIPS R10000	2-3	2-3	11-18
UltraSPARC III	4	4	24
Cyrix 5x86 and 6x86	4-9	4-9	24-34
Alpha21264	4	4	15
Athlon K6-III	3	3	20

Table 3.5: *Latencies of some floating-point instructions in double-precision for various processors, after [87, 285, 286, 101].*

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,[6] 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 current processors. Those figures clearly show that for the moment, division is much slower than multiplication, so it is frequently preferable to use polynomial approximations.[7] This might change in the future: studies by Oberman and Flynn [246, 247] tend to show that fast division units could contribute to better performance in many areas.[8] As a consequence, future processors might offer faster divisions.

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

[7] Unless some parallelism is available in the processor being used or the circuit being designed. For instance, as pointed out by Koren and Zinaty [191], 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.

[8] 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.

	frac1	frac2	frac3
worst-case error	$0.3110887e-14$	$0.1227446e-14$	$0.1486132e-14$
average error	$0.3378607e-15$	$0.1847124e-15$	$0.2050626e-15$

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*.*

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

$$\text{frac1}\ (x) = \frac{3 - 9x + 15x^2 - 12x^3 + 7x^4}{1 - x + x^2},$$

$$\text{frac2}\ (x) = 3 - 5x + 7x^2 - \frac{x}{1 - x + x^2},$$

$$\text{frac3}\ (x) = 3 + x \times \frac{-6 + 12x - 12x^2 + 7x^3}{1 - x + x^2},$$

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 [68]. It has been used by Hamada [154]. 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.

3.7 Actual Computation of Approximations

3.7.1 Getting "general" 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 [57] readily computes minimax or Chebyshev approximations. 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
-------------------------------------------------------
> 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 + (
```

```
.42170301312917803944060417427007517003 36

+ .27997648897921345565571846527282751555 2 x) x) x

.00054479156433258976434258817641527074 5
```

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}.

3.7.2 Getting approximations with special constraints

It is sometimes desirable to have polynomial or rational approximations with a particular form (for instance, $x+x^3p(x^2)$ for the sine function: this preserves symmetry and makes the number of multiplications used for evaluation smaller), or with a fixed value at zero, or that are provably monotone. Such approximations have been studied by Dunham [115, 117, 118, 120, 121, 122]. Moreover, although the methods we presented in this chapter help to bound the absolute approximation error, one may be interested in bounding the relative error. Let us examine an example, inspired by reference [120].

Example 3 (Sine function on $[0, \pi/8]$) *Assume that we want to approximate the sine function on $[0, \pi/8]$, with a relative error bounded by ϵ, by a polynomial of the form:*

$$x + a_3x^3 + a_5x^5 + \cdots + a_{2n+1}x^{2n+1} = x + x^3p(x^2),$$

where $p(x) = a_3 + a_5x + a_7x^2 + \cdots + a_{2n+3}x^n$. We want

$$\left| \frac{\sin(x) - x - x^3p(x^2)}{\sin(x)} \right| \leq \epsilon. \tag{3.5}$$

This is equivalent to

$$\left| \frac{\frac{\sin(x)}{x^3} - \frac{1}{x^2} - p(x^2)}{\frac{\sin(x)}{x^3}} \right| \leq \epsilon.$$

Now, define $X = x^2$. Equation (3.5) is equivalent to:

$$\left| \frac{\frac{\sin(\sqrt{X})}{X^{3/2}} - \frac{1}{X} - p(X)}{\frac{\sin(\sqrt{X})}{X^{3/2}}} \right| \leq \epsilon. \tag{3.6}$$

Therefore our problem is reduced to finding a minimax polynomial approximation $p(X)$ *to*

$$\frac{\sin(\sqrt{X})}{X^{3/2}} - \frac{1}{X}$$

with a weight function $X^{3/2}/\sin(\sqrt{X})$ *for* $X \in [0, \pi^2/64]$. *Using the Taylor expansion:*

$$\frac{\sin(\sqrt{X})}{X^{3/2}} - \frac{1}{X} = -\frac{1}{6} + \frac{X}{120} - \frac{X^2}{5040} + \cdots + \frac{(-1)^{2n+1}X^n}{(2n+3)!} + \cdots,$$

we can find $p(X)$ *using the Maple function*

```
minimax(-1/6+X/120-X^2/5040+X^3/362880-X^4/39916800
+X^5/6227020800,X=0..Pi^2/64,[2,0],
((sqrt(X^3))/sin(sqrt(X))),'err');
```

which returns the result

```
-.16666666480509 + (.0083332602856 - .000197596738 X) X.
```

Maple also gives the approximation error:

```
> err;
                      -10
           .14363 10    .
```

Therefore the following polynomial approximates $\sin(x)$ *on* $[0, \pi/8]$ *with a relative error less than* 2×10^{-11},

$$x - 0.16666666480509x^3 + 0.0083332602856x^5 - 0.000197596738x^7.$$

It is possible to recompute the approximation error as follows. The Maple command

```
> infnorm((x - 0.16666666480509*x^3+0.0083332602856*x^5-
0.000197596738*x^7-sin(x))/sin(x),x=0..Pi/8);
```

gives

```
                                      -10
                  .143663703077561 10    .
```

For software implementations, when the polynomial or rational approximations are to be evaluated in a precision that is not higher than the target precision, it is important to get approximations with coefficients that are machine numbers.[9] A similar method can be used again, as follows. Compute the approximation with extra precision, round the leading coefficient to the closest machine number, then recompute an approximation where you impose the leading coefficient to be the previously rounded one.

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

Example 4 (Computation of 2^x) *For instance, 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 single-precision arithmetic. We first compute, in extended-precision arithmetic the minimax approximation:*

```
0.99999999927558511254956761285
+(0.69314725465976987804798257293
+(0.24021466637817386753346917l927
+0.05610900238939350523988382289l8*x)*x)*x.
```

After this, we impose the coefficient of degree 0 to be $a_0 = 1$, and the coefficient of degree 1 to be the single-precision number that is closest to the coefficient of degree 1 of the previously computed approximation; that is,

$$a_1 = 0.6931472420692443847656l5 = \frac{11629081}{2^{24}}.$$

Now if ϵ is the approximation error of the new approximation we wish to compute, and if we define $p(x)$ to be the degree-1 polynomial $a_2 + a_3x$, we want:

$$\left|(2^x - 1 - a_1x) - x^2p(x)\right| \leq \epsilon.$$

This is equivalent to

$$\left|\frac{\dfrac{2^x - 1 - a_1x}{x^2} - p(x)}{\dfrac{1}{x^2}}\right| \leq \epsilon.$$

Therefore it suffices to compute a degree-1 minimax approximation of

$$\frac{2^x - 1 - a_1x}{x^2}$$

with a weight function x^2. This gives:

$$\begin{aligned} a_2 &= 0.24021510205 \\ a_3 &= 0.05610760819, \end{aligned}$$

and the approximation error is roughly equal to 10^{-10}.

The methods we have given so far in this section frequently work well, and yet, in general, they do not give the *best* polynomial (or rational) approximation among the ones that satisfy the constraints.

Let us now describe how to get such best approximations. The following method was suggested by Brisebarre, Muller and Tisserand [49]. 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 [49] consists in first finding a *polytope* (i.e., a bounded polyedron) where p^* necessarily lies,[10] 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 [6, 77, 308], 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[11] constraints[12]

$$f(x_j) - K \leq p_0^* + p_1^* x_j + p_2^* x_j^2 + \cdots + p_n^* x_j^n \leq f(x_j) + K, \forall j. \tag{3.7}$$

If $K < \epsilon$, then the polytope defined by (3.7) 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 [49] for more details.

3.8 Algorithms and Architectures for the Evaluation of Polynomials

In the previous sections, 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.

[10] 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.

[11] Linear in the coefficients p_i^*.

[12] In practice, we use slightly different constraints: we modify (3.7) so that we can work with rational numbers only.

Using existing operators for addition and multiplication, we can only give some 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;
```

  or, even worse:

```
a_4*power(x,4)+a_3*power(x,3)+a_2*power(x,2)
+a_1*x+a_0;
```

- in general, if the coefficients of the polynomial do not satisfy a specific property (e.g., a simple factorization) that could help to accelerate the computation, it is advisable to use *Horner's scheme* :

```
(((a_4*x+a_3)*x+a_2)*x+a_1)*x+a_0.
```

Although commonly attributed to Horner, this method was used previously by Newton [187].

Some processors (for instance the Power PC, or the Itanium) have a "fused multiply-add" (FMA, fused MAC) instruction; that is, an expression of the form $\pm a \times x + b$ can be evaluated just with one instruction, and there is only *one* rounding error at the end. These properties can be used to quickly and accurately evaluate a polynomial. If the depth of the pipeline that performs the fused multiply-add is small, then Horner's scheme is interesting. Otherwise, Estrin's method (see Section 3.8.2) may become attractive.

If the degree of the polynomial is large,[13] one can use a method called "adaptation of coefficients," that was analyzed by Knuth [187]. 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 9 (Knuth) *Let $u(x)$ be a degree-n polynomial*

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

Let $m = \lfloor n/2 \rfloor - 1$. There exist parameters $c, \alpha_1, \alpha_2, \ldots, \alpha_m$ and $\beta_1, \beta_2, \ldots, \beta_m$ such that $u(x)$ can be evaluated using at most $\lfloor n/2 \rfloor + 2$ multiplications and n additions by performing the following calculations:

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

[13] Which sometimes occurs, see for instance [83], page 267.

The preceding expression that gives $u(x)$ as a function of the parameters c, $\alpha_1, \alpha_2, \ldots, \alpha_m$, and $\beta_1, \beta_2, \ldots, \beta_m$ leads to a nonlinear system of equations. In this system, the number of unknown variables is 1 or 2 plus the number of equations; therefore, in general, there are solutions for most values of c. If $n = 8$ and if we choose $c = 1$, the system of equations becomes:

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

Computing the coefficients c, $\alpha_1, \alpha_2, \ldots, \alpha_m$, and $\beta_1, \beta_2, \ldots, \beta_m$ 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 $n = 8$, $u_i = 1/i$ 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. More details on polynomial evaluation can be found in [187, 188].

When designing specific hardware, it may be possible to use some algorithms and architectures for evaluating polynomials that have been proposed in the past. Let us examine two such solutions.

3.8.1 The E-method

The *E-method*, introduced by M.D. Ercegovac in [128, 129], 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.

$$\begin{bmatrix} 1 & -x & 0 & & \cdots & & 0 \\ 0 & 1 & -x & 0 & & \cdots & 0 \\ 0 & 0 & 1 & -x & 0 & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \ddots & \ddots & 0 \\ & & & & & 1 & -x \\ 0 & & \cdots & & & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \\ \vdots \\ \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \\ \vdots \\ \\ p_{n-1} \\ p_n \end{bmatrix}. \tag{3.8}$$

The radix-2 E-method consists of solving this linear system by means of the basic recursion

$$\begin{aligned} w^{(0)} &= [p_0, p_1, \ldots, p_n]^t \\ w^{(j)} &= 2 \times \left[w^{(j-1)} - A d^{(j-1)} \right], \end{aligned} \tag{3.9}$$

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 [129], the following selection function (a form of rounding) is proposed,

$$s(x) = \begin{cases} \text{sign}\,(x) \times \lfloor |x + 1/2| \rfloor, & \text{if } |x| \leq 1 \\ \text{sign}\,(x) \times \lfloor |x| \rfloor, & \text{otherwise,} \end{cases} \tag{3.10}$$

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

$$\begin{cases} \xi = \frac{1}{2}(1 + \Delta) \\ 0 < \Delta < 1 \\ \alpha \leq \frac{1}{4}(1 - \Delta). \end{cases} \tag{3.11}$$

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 [129] that allow us to compute $p(x)$ for any x and p.

3.8.2 Estrin's method

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

$$a_7 x^7 + a_6 x^6 + \cdots + a_1 x + a_0.$$

If we are able to perform multiplications and accumulations in parallel (or in a pipe-lined fashion), we can use Estrin's algorithm [187].

Algorithm 5 (Estrin)

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

Perform the following steps:

1. *In parallel, compute* $X^{(1)} = x^2$, $a_3^{(1)} = a_7 x + a_6$, $a_2^{(1)} = a_5 x + a_4$, $a_1^{(1)} = a_3 x + a_2$, *and* $a_0^{(1)} = a_1 x + a_0$,
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* $p(x) = a_1^{(2)} X^{(2)} + a_0^{(2)}$.

This algorithm, given for degree-7 polynomials for the sake of simplicity, can be easily extended to polynomials of any degrees.

Estrin's method is used for instance in some of INTEL's elementary function programs for the Itanium [83]. The basic idea behind the *Polynomier* [123] consists of performing the multiplications and accumulations required by Estrin's method in pipeline, using a modified Braun's multiplier [37, 174]. A prototype of the Polynomier was designed in the Swiss Federal Institute of Technology of Lausanne [81]. Estrin's method becomes very interesting to consider when programming a polynomial evaluation on a machine with a pipelined fused multiply-add. For instance, implementation on the Intel Itanium circuit is discussed in [83] (pp. 62 and seq.).

3.9 Evaluation Error Assuming Horner's Scheme is Used

So far, 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. These errors will result in a final evaluation error. We therefore have to find sharp bounds on that evaluation error. We assume here that the computations are performed using 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 also assume that the polynomial is evaluated using Horner's scheme. We wish to tightly bound the largest possible evaluation error, for $x \in [x_{\min}, x_{\max}]$.

There are some interesting results in the literature, for instance (see Higham's book [165] for more information),

Theorem 10 ([165], Corollary 3 of [24]) *The error in the evaluation of*

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

using Horner's scheme is bounded by

$$2 \times \mathit{ulp}\,(1) \times n \times \sum_{i=0}^{n} |a_i x^i| + O(\mathit{ulp}\,(1)^2).$$

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 error-prone, that I strongly recommend that these calculations be automated. In this section, 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 function `ulp` given in Section 2.1.8. 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).

3.9.1 Evaluation using floating-point additions and multiplications

Definitions

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 mode is selected. 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

$$\begin{cases} P[i-1] = S[i-1] \otimes x \\ S[i-1] = P[i] \oplus a_{i-1}, \end{cases}$$

where $\oplus$ and $\otimes$ are the floating-point addition and multiplication, respectively, 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

$$P[i] = S[i] \otimes x;$$

- $\epsilon[i-1]$ is an upper bound on the error due to the floating-point addition

$$S[i-1] = P[i] \oplus a_{i-1}.$$

Now, define $\text{err}\,[i]$ as an upper bound on $|S^*[i] - S[i]|$. We wish to compute the value of $\text{err}\,[0]$: this is the bound on the final evaluation error we are looking for. We will compute it iteratively, starting from $\text{err}\,[n] = 0$.

Computing the error bounds iteratively

We first start from the straightforward values $S_{min}[n] = S_{max}[n] = a_n$, and we define $\text{err}\,[n] = 0$. Now, assume that we know $S_{min}[i]$, $S_{max}[i]$ and $\text{err}\,[i]$. Let us see how to deduce $S_{min}[i-1]$, $S_{max}[i-1]$ and $\text{err}\,[i-1]$. First, we obviously find (this is the usual interval multiplication)

$$\hat{P}_{min}[i] = \min\left\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\right\}$$

and

$$\hat{P}_{max}[i] = \max\left\{S_{min}[i]x_{min}, S_{min}[i]x_{max}, S_{max}[i]x_{min}, S_{max}[i]x_{max}\right\}.$$

Since we use correctly rounded multiplication, in round-to-nearest mode, the rounding error that occurs when computing $S[i] \otimes x$ is upper-bounded by

$$\frac{1}{2}\,\text{ulp}\,(S[i]x).$$

Since $\text{ulp}\,(t)$ 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

$$\delta[i] = \frac{1}{2}\,\text{ulp}\,\left(\max\left\{|\hat{P}_{min}[i]|, |\hat{P}_{max}[i]|\right\}\right).$$

We immediately deduce the following lower and upper bounds on $P[i]$:

$$\begin{cases} P_{min}[i] = \hat{P}_{min}[i] - \delta[i] \\ P_{max}[i] = \hat{P}_{max}[i] + \delta[i]. \end{cases}$$

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 $S[i-1] = P[i] \oplus a_{i-1}$:

$$\epsilon[i-1] = \frac{1}{2}\,\text{ulp}\,\left(\max\left\{|\hat{S}_{min}[i-1]|, |\hat{S}_{max}[i-1]|\right\}\right),$$

which gives the following lower and upper bounds on $S[i-1]$:

$$\begin{cases} S_{min}[i-1] = \hat{S}_{min}[i-1] - \epsilon[i-1] \\ S_{max}[i-1] = \hat{S}_{max}[i-1] + \epsilon[i-1]. \end{cases}$$

We now have all the information we need to compute err $[i-1]$:

$$\text{err}\,[i-1] = \text{err}\,[i] \max\{|x_{min}|, |x_{max}|\} + \delta[i] + \epsilon[i-1].$$

The following Maple program uses these relations for deducing 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.

```
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(max(abs(pminhat[i]),abs(pmaxhat[i])));
  pmin[i] := pminhat[i]-delta[i];
  pmax[i] := pmaxhat[i]+delta[i];
  sminhat[i-1] := pmin[i]+a[i-1];
  smaxhat[i-1] := pmax[i]+a[i-1];
  epsilon[i-1] := 0.5*ulp(max(abs(sminhat[i-1]),
                      abs(smaxhat[i-1])));
  smin[i-1] := sminhat[i-1]-epsilon[i-1];
  smax[i-1] := smaxhat[i-1]+epsilon[i-1];
  err[i-1] := err[i]*max(abs(xmin),abs(xmax))
                +epsilon[i-1]+delta[i]
 od;
err[0];
end;
```

Two examples, and some refinement

Consider the degree-5 polynomial

$$\begin{aligned} p(x) = 1 &+ \frac{2251799813685251}{2251799813685248}\,x + \frac{2251799813675657}{4503599627370496}\,x^2 \\ &+ \frac{3002399772526855}{18014398509481984}\,x^3 + \frac{3002379076408669}{72057594037927936}\,x^4 \\ &+ \frac{4822644803220247}{576460752303423488}\,x^5. \end{aligned}$$

That polynomial, whose coefficients are exactly representable in the IEEE-754 double-precision format, is an approximation to the exponential function in $[0, 1/128]$.

Our Maple program makes it possible to get a bound on the error committed when evaluating $p(x)$ in double-precision arithmetic, for $x \in [0, 1/128]$. This is done (assuming that $a[i]$ contains the degree-i coefficient of p) by typing

```
> Errevalpol(a,5,0,1/128);
```

which returns 1.128×10^{-16}. It is worth noticing that in this case the error bound given by procedure `Errevalpol` is rather tight: by computing $p(x)$ in double-precision for 200 randomly chosen values of x and comparing with the exact values, the largest obtained error was 1.114×10^{-16}. Since in the considered domain, $\mathrm{ulp}\,(p(x)) = \mathrm{ulp}\,(\exp(x)) = 2^{-52}$, we deduce that we evaluate p in $[0, 1/128]$ with error less than $1.128 \times 10^{-16} \times 2^{52} = 0.508$ ulps. We can easily bound the evaluation error obtained if we now assume that an internal extended-precision format is available (as on Intel processors). This is done by redefining function `ulp` of Section 2.1.8 to take into account the new format, and by adding $1/2$ ulp *of the double-precision format* (that is, in the present example, 2^{-53}) to the obtained error, to take into account the final rounding of the obtained double-extended precision result to the target double-precision format. This leads to an evaluation of p with an error less than 0.50025 ulps.

Now, let us switch to another example. Consider, for $x \in [0, 1]$, the polynomial

$$\begin{aligned} q(x) = & \frac{4502715367124429}{4503599627370496} - \frac{5094120834338589}{9007199254740992}\, x \\ & + \frac{3943097548915637}{4503599627370496}\, x^2 - \frac{272563672039763}{562949953421312}\, x^3 \\ & + \frac{6289926120511169}{36028797018963968}\, x^4. \end{aligned}$$

It is a (rather poor, but this does not matter here) approximation to $x! = \Gamma(x+1)$ in $[0, 1]$. Using function `Errevalpol`, we get an error bound equal to 4.025×10^{-16} (i.e., 3.625 ulps), whereas the largest error we have actually obtained through experiments is around 1.495 ulps, which is 2.4 times less.

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 the previous example, 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 sub-intervals, to use `Errevalpol` in each sub-domain, 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 64 sub-intervals, by calling `RefinedErrPol(a,4,0,1,64)`, we get a better error bound: 2.93×10^{-16}, that is, around 2.64 ulps.

3.9.2 Evaluation using fused multiply-accumulate instructions

Definitions

Let us now assume that on the target architecture a fused-multiply accumulate instruction is available, and that we use that instruction to implement the polynomial evaluation by Horner's scheme. This makes it possible to evaluate an expression $ax + b$ with one final rounding only.

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 $S^*[i]$ using:

$$S[i-1] = (S[i]x + a_{i-1}) \text{ rounded to nearest},$$

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 3.9.1, err $[i]$ is an upper bound on $|S^*[i] - S[i]|$. We wish to compute err $[0]$: this is the final evaluation error we are looking for. We will compute it iteratively, starting from err $[n] = 0$.

Iteratively computing the evaluation error

The iterative process that gives err $[0]$ is very similar to the one described in Section 3.9.1. We first start from the straightforward values: $S_{min}[n] = S_{max}[n] = a_n$ and err $[n] = 0$.

Now, assume that we know $S_{min}[i]$, $S_{max}[i]$ and err $[i]$. Let us see how to deduce $S_{min}[i-1]$, $S_{max}[i-1]$ and 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

$$\epsilon[i-1] = \frac{1}{2}\,\text{ulp}\,\left(\max\left\{|\hat{S}_{min}[i-1]|, |\hat{S}_{max}[i-1]|\right\}\right),$$

which gives the following lower and upper bounds

$$\begin{cases} S_{min}[i-1] = \hat{S}_{min}[i-1] - \epsilon[i-1] \\ S_{max}[i-1] = \hat{S}_{max}[i-1] + \epsilon[i-1]. \end{cases}$$

We now have all the information we need to compute err $[i-1]$:

$$\text{err}\,[i-1] = \text{err}\,[i]\max\left\{|x_{min}|, |x_{max}|\right\} + \epsilon[i-1].$$

The following Maple program uses these relations for deducing 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.

```
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(max(abs(sminhat[i-1]),
                  abs(smaxhat[i-1])));
  smin[i-1] := sminhat[i-1]-epsilon[i-1];
  smax[i-1] := smaxhat[i-1]+epsilon[i-1];
  err[i-1] := err[i]*max(abs(xmin),abs(xmax))+epsilon[i-1]
 od;
err[0];
end;
```

Examples

Let us first consider the degree-5 polynomial $p(x)$ of Section 3.9.1, still assuming double-precision arithmetic, but using FMAs instead of floating-point additions and multiplications. The error bound we get, for $x \in [0, 1/128]$, using procedure `ErrevalpolFMA` is 1.119×10^{-16}. Since, for $x \in [0, 1/128]$, $\mathrm{ulp}(x) = \mathrm{ulp}(1)$, this error bound is 0.504 ulps.

If we evaluate $p(x)$ in double-extended precision, and round the obtained final result to double-precision, the error bound becomes $1.110770 \times 10^{-16} \approx 0.50027$ ulps.

Now, let us focus on the degree-4 polynomial $q(x)$ of Section 3.9.1, with double-precision arithmetic. We get an error bound equal to $2.498 \times 10^{-16} = 2.25$ ulps, and, using a refinement technique similar to the one suggested in Section 3.9.1 (by dividing the input interval into 256 sub-intervals), the error bound becomes 1.752 ulps. The largest error found (through evaluation of q for 400 random input values very close to 1) is 1.567 ulps.

The Gappa software[14] developed by Melquiond automatically computes bounds on the roundoff error that occurs during some numerical computations such as the ones considered in this section, and generates formal proofs of these bounds.

3.10 Miscellaneous

Polynomial and rational approximations of functions have been widely studied [62, 64, 65, 99, 116, 125, 141, 142, 146, 160, 191, 265]. Good references on approximation are the books by Cheney [59], Rice [267], Rivlin [268] and Laurent [201]. Some of the ideas presented in this chapter can be extended to complex elementary functions. Braune [38] and Krämer [193] suggested algorithms that evaluate standard functions and inverse standard functions for real and complex point and interval arguments with dynamic accuracy. Midy and Yakovlev [230] and Hull et al. [172] suggested algorithms for computing elementary functions of a complex variable. The CELEFUNT package, designed by W.J. Cody [72], is a collection of test programs for the complex floating-point elementary functions. Minimax approximation by polynomials on the unit circle is considered in [16, 313], with applications to digital filtering. When evaluating a given polynomial may lead to underflow or overflow, there exist scaling procedures [155] that prevent overflow and reduce the probability of occurrence of underflow. Also, when high precision is not at stake, one can try to use the multimedia SIMD features available on some recent processors to accelerate polynomial evaluation, as suggested by Bandera et al. [23]. In his PhD dissertation, Liddicoat [215] presents an arithmetic operator that evaluates polynomial approximations to functions.

[14] `http://lipforge.ens-lyon.fr/www/gappa/`.

Chapter 4

Table-Based Methods

4.1 Introduction

Evaluating a function by approximating it in a rather large domain using the techniques presented in Chapter 3 may require polynomial or rational functions of large degrees. This may lead to long delays of computation, and this may also make the numerical error control difficult. A natural way to deal with this problem is to split the interval where the function is to be approximated into several smaller subintervals. It suffices to store in a table, for each subinterval, the coefficients of a low-degree approximation that is valid in that interval. Such a method is not new (a PDP-9 implementation is reported in [10]), but it may become very attractive nowadays, since memory is less and less expensive. And yet, for software-based implementation, the use of large tables may increase the probability of a cache miss. See [100, 101] for a discussion on this problem. This is why some implementations of some functions do not use tables at all (for instance the arctangent function described in [83, 159] uses a polynomial of degree 47 and no table).

Let us consider the following example.

Example 5 (Sine function on $[0, \pi/4]$) *We wish to approximate the sine function in $[0, \pi/4]$, with an error less than 10^{-8}. Table 4.1 shows that if we do not split the interval $[0, \pi/4]$, and use one polynomial approximation only, then a polynomial of degree 6 is necessary. Table 4.2 shows that if we split the interval into two subintervals of equal size, then degree-5 polynomial approximations suffice. Table 4.3 shows that if we split $[0, \pi/4]$ into 4 subintervals, then approximations of degree 4 suffice.*

This example shows that a significant amount of computation time can be saved by splitting the original domain. Many compromises between the amount of computation and the amount of storage can be found. Much care is needed at the boundaries of the subdomains if we wish to preserve properties such as monotonicity. Tables 4.4 and 4.5 show for various functions that reducing the size of the interval where a function is approximated allows the use of a

Interval	Degree	Error
$[0, \pi/4]$	5	0.609×10^{-7}
	6	0.410×10^{-8}
	7	0.418×10^{-10}

Table 4.1: *Minimax approximation to* $\sin(x), x \in [0, \pi/4]$*, using one polynomial. The errors given here are absolute errors.*

Interval	Degree	Error
$[0, \pi/8]$	4	0.148×10^{-6}
	5	0.486×10^{-9}
	6	0.342×10^{-10}
$[\pi/8, \pi/4]$	4	0.126×10^{-6}
	5	0.138×10^{-8}
	6	0.289×10^{-10}

Table 4.2: *Minimax approximation to* $\sin(x), x \in [0, \pi/4]$*, using two polynomials. The errors given here are absolute errors.*

Interval	Degree	Error
$[0, \pi/16]$	3	0.478×10^{-7}
	4	0.472×10^{-8}
	5	0.382×10^{-11}
$[\pi/16, \pi/8]$	3	0.140×10^{-6}
	4	0.454×10^{-8}
	5	0.113×10^{-10}
$[\pi/8, 3\pi/16]$	3	0.228×10^{-6}
	4	0.418×10^{-8}
	5	0.183×10^{-10}
$[3\pi/16, \pi/4]$	3	0.307×10^{-6}
	4	0.367×10^{-8}
	5	0.246×10^{-10}

Table 4.3: *Minimax approximation to* $\sin(x), x \in [0, \pi/4]$*, using four polynomials. The errors given here are absolute errors.*

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 4.4: *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	exp	$\ln(1+x)$
10	19	16	15
1	6	5	5
0.1	3	2	3
0.01	1	1	1

Table 4.5: *Degrees of the minimax polynomial approximations that are required to approximate some functions with error less than* 10^{-5} *on the interval* $[0, a]$. *When* a *becomes small, a very low degree suffices.*

polynomial of very small degree. With the most common 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 instance, when computing the exponential function in a domain of the form $[0, a]$, with equally sized sub-intervals, it suffices to have an approximation valid in the first interval. In the subinterval $[a_k, a_{k+1}]$, the exponential of x is e^{a_k} times the exponential of $x - a_k$, and $x - a_k$ obviously belongs to the first subinterval. Similar tricks can be obtained for the trigonometric functions and the logarithms.

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 implemented using

(or simulating [102]) a precision that is somewhat larger than the target precision; by simply tabulating the function in the target precision, an error that can be close to $1/2$ ulp is 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. This is the case on Intel processors that offer, since the 8087, an internal 80-bit format. Tang's "table-driven" algorithms [304, 305, 306, 307] (see Section 4.2) 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 4.3) 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 4.4.1 or the bipartite method, presented in Section 4.4.4. Such methods require hardware implementation.

4.2 Table-Driven Algorithms

In [304, 305, 306, 307], 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 9), 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 [304]. After this, we briefly give Tang's guidelines for $\ln(x)$ and $\sin(x)$ [306].

4.2.1 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 [304] suggests first reducing the input argument to a value r in the interval

$$\left[-\frac{\ln(2)}{64}, +\frac{\ln(2)}{64}\right],$$

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

$$x = (32m + j)\frac{\ln(2)}{32} + r, \quad 0 \le j \le 31. \tag{4.1}$$

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 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:

$$r_1 = x - N \times L^{\text{left}}$$

and

$$r_2 = -N \times L^{\text{right}}.$$

The values m and j of 4.1 are $m = N_1/32$ and $j = N_2$. An analysis of this reduction method is given in reference [139]. Other reduction methods, that may be more accurate for large values of x, are presented in Chapter 9.

[1] For single-precision, see [304].

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

$$\exp(x) = 2^m \times \left(s_j^{\text{left}} + \left(s_j^{\text{right}} + S_j \times p(r)\right)\right).$$

4.2.2 $\ln(x)$ 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 given in Table 4.6.

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.

Degree	Error	Polynomial
3	0.48×10^{-13}	r $+0.0833339964r^3$
5	0.10×10^{-18}	r $+0.0833333333290521r^3$ $+0.01250020282r^5$
7	0.27×10^{-24}	r $+0.083333333333335805r^3$ $+0.0124999999978878r^5$ $+0.002232197165r^7$
9	0.80×10^{-30}	r $+0.083333333333333333333200498r^3$ $+0.012500000000000017673r^5$ $+0.0022321428563634129r^7$ $+0.000434041799769r^9$

Table 4.6: *Approximations to* $\ln((1+r/2)/(1-r/2))$ *on* $[0, 1/128]$.

If a larger accuracy is required, one can use more breakpoints; with 256 breakpoints and a polynomial of degree 7, the approximation error will be 0.1×10^{-29}. Using 512 breakpoints, the approximation error will be 0.2×10^{-32} with a polynomial of degree 7, and 0.92×10^{-40} with a polynomial of degree 9.

4.2.3 $\sin(x)$ on $[0, \pi/4]$

reduction: If $|x|$ is very small (the threshold chosen by Tang in [306] 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 $\sin(r) - r$ and $\cos(r) - 1$, for instance, using one of the polynomials given in Table 4.7 for the first function, and one of the polynomials given in Table 4.8 for the second one.

reconstruction: We reconstruct $\sin(x)$ using:

$$\sin(x) = \sin(c_{jk})\cos(r) + \cos(c_{jk})\sin(r).$$

4.3 Gal's Accurate Tables Method

This method is due to Gal [147], 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 [148]

Degree	Error	Polynomial
3	0.33×10^{-10}	$-0.1666596r^3$
5	0.15×10^{-15}	$-0.16666666656924r^3$ $+0.008333044883r^5$
7	0.45×10^{-21}	$-0.16666666666666602381875r^3$ $+0.00833333329900320r^5$ $-0.0001984071815851r^7$
9	0.10×10^{-26}	$-0.16666666666666666666421r^3$ $+0.00833333333333312907r^5$ $-0.0001984126983563939r^7$ $+0.00000275566861r^9$

Table 4.7: *Approximations to* $\sin(r) - r$ *on* $[-1/32, 1/32]$.

Degree	Error	Polynomial
2	0.68×10^{-8}	$-0.49996629r^2$
4	0.50×10^{-13}	$-0.49999999942942r^2$ $+0.04166477827r^4$
6	0.21×10^{-18}	$-0.4999999999999576r^2$ $+0.041666666640330r^4$ $-0.0013888423656r^6$
8	0.54×10^{-24}	$-0.49999999999999999999825089r^2$ $+0.041666666666666492647r^4$ $-0.0013888888883507292r^6$ $+0.0000248009314r^8$
10	0.98×10^{-30}	$-0.49999999999999999999999995425696r^2$ $+0.04166666666666666666660027496r^4$ $-0.00138888888888888885759632r^6$ $+0.000024801587295150563 6r^8$ $-0.000000275567182072r^{10}$

Table 4.8: *Approximations to* $\cos(r) - 1$ *on* $[-1/32, 1/32]$.

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 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]

[2]We call *target accuracy* the accuracy of the number system used for representing the results.

[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 10.

By doing this, we simulate a *larger accuracy*. Let us consider the following example.

Example 6 (computation of the exponential function) *Assume that we use a base-*10 *computer with* 4*-digit-mantissas, 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* $x - \frac{i+1/2}{10}$. *The values stored in the table are:*

x	e^x	*Value Stored*	$\mid$ *Error* $\mid$
0.55	$1.733253\cdots$	1.733	2.5×10^{-4}
0.65	$1.915540\cdots$	1.916	4.6×10^{-4}
0.75	$2.117000\cdots$	2.117	1.7×10^{-8}
0.85	$2.339646\cdots$	2.340	3.5×10^{-4}
0.95	$2.585709\cdots$	2.586	2.9×10^{-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 *digits);*
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	1.2×10^{-6}
0.6518	$1.91899190\cdots$	1.919	8.1×10^{-6}
0.7500	$2.11700001\cdots$	2.117	1.7×10^{-8}
0.8493	$2.33800967\cdots$	2.338	9.6×10^{-6}
0.9505	$2.58700283\cdots$	2.587	2.8×10^{-6}

Now, the rounding error becomes 9.6×10^{-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* 50 *times for the worst case.*

Now let us study Gal and Bachelis' algorithm for computing sines and cosines in IEEE-754 double-precision floating-point arithmetic [148]. We assume that a range reduction has been performed (see Chapter 9), 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 $z = (u - X_i) + du$, 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 percent of the tested cases. This shows that for software implementation, a careful programming of the accurate tables method is an interesting choice.[4]

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[5] at the point X_i is very close to a machine number. Let us assume that we use an n-bit-mantissa, 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 n 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 need the same assumption in Chapter 10 for studying the possibility of always computing exactly rounded results). We also assume that when we need to tabulate

[4] 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.

[5] We need one function for computing exponentials, two for computing sines or cosines.

several functions $f_1, f_2, \ldots, f_k$, the bits of $f_1(x)$, $f_2(x)$, ..., $f_k(x)$ can be viewed as "independent."

Using our assumptions, one can easily see that the "probability" of having p 0s or p 1s after position n in $f_1(x)$, $f_2(x)$, ..., $f_k(x)$ is 2^{-kp+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 test around $q \times 2^{kp-k}$ 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 $p = 11$, we have to test a number of values whose order of magnitude is 2×10^8;

- 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 $2^{n-\alpha}\epsilon$. Therefore, to make sure that there exist such values X_i, 2^{kp-k} must be small compared to $2^{n-\alpha}\epsilon$.

Luther [220] described a fast method for obtaining highly accurate tables by using Bresenham's algorithm. Stehlé and Zimmermann recently suggested improvements to the accurate tables method [293]. Using algorithms originally designed for finding worst cases of the table maker's dilemma (see Chapter 10 and references [206, 208, 211, 292]), they generate values X_i that are extremely good values, faster than when using Gal's original searching method.

4.4 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 [324] suggested using a specialized hardware for implementing a table-based method. To evaluate the elementary functions in the IEEE-754 double-precision floating-point format (see Chapter 2), they use a *rectangular multiplier*, more precisely, a 16×56-bit multiplier[6] that truncates the final result to 56 bits. As they point out, such multipliers already exist in actual floating-point circuits: the Cyrix 83D87 coprocessor had a 17×69 bit multiply and add array [45]. A rectangular multiplier is faster than a full double-precision 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 double-precision multiplication. Let us now examine some of Wong and Goto's algorithms.

[6] For most of their algorithms, a 56×12 bit rectangular multiplier suffices.

4.4.1 Wong and Goto's algorithm for computing logarithms

Assume we wish to compute the logarithm of a normalized IEEE-754 double-precision 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 cancellation*.[7] 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[8]:

$$\frac{1}{2} \leq m < 2.$$

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

$$[m]_{1-3} = 0.m_1m_2m_3000\ldots.$$

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_1K_2m is even closer to 1. After this our problem is reduced to evaluate the logarithm of (K_1K_2m). 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 [324]).

[7] A catastrophic cancellation 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 2), if x and y are floating-point numbers in the same format such that $x/2 \leq y \leq 2x$, then the subtraction $y - x$ is exactly performed on any machine compliant with the IEEE-754 standard.

[8] But we must keep in mind that $1/2 \leq m < 1$ if $exponent = 0$ only; otherwise, we could have a catastrophic cancellation for $exponent = 1$ and m close to $1/2$.

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 double-precision 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 $a = 0.00000000a_9a_{10}\cdots a_{18}$:

$$a_9a_{10}\cdots a_{18} = \begin{cases} \alpha_9\alpha_{10}\cdots\alpha_{17}\alpha_{18}+1 & \text{if } \alpha_9 = 1 \\ \alpha_9\alpha_{10}\cdots\alpha_{17}0 & \text{otherwise,} \end{cases} \tag{4.2}$$

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 (4.2)) $0 \le \beta < 2^{-16}$. 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

$$b_9b_{17}\cdots b_{26} = \begin{cases} \beta_{17}\beta_{18}\beta_{19}\cdots\beta_{25}\beta_{26}+1 & \text{if } \beta_{17} = 1 \\ \beta_{17}\beta_{18}\beta_{19}\cdots\beta_{25}0 & \text{otherwise,} \end{cases} \tag{4.3}$$

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.

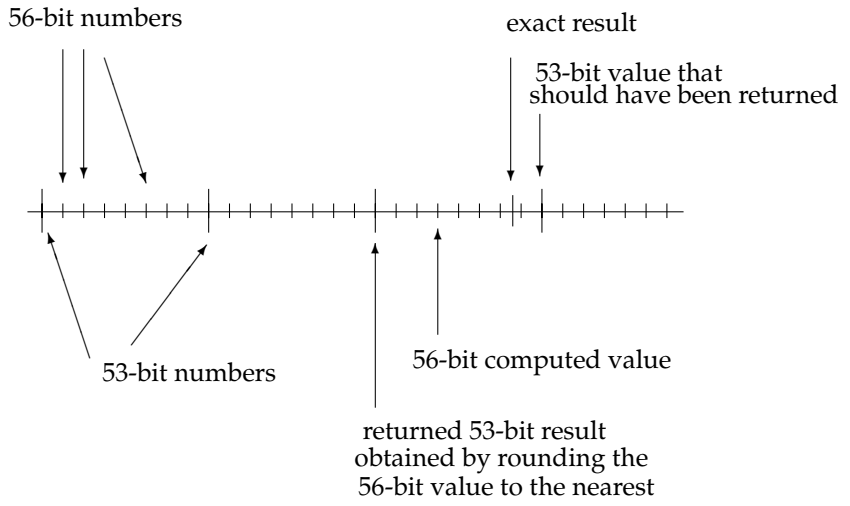


Figure 4.1: *An incorrectly rounded result deduced from a 56-bit value that is within 0.5 ULPs from the exact result. We assume that rounding to the nearest was desired.*

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 [324] 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-mantissa bit IEEE double-precision 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 4.1 illustrates this. Getting correctly rounded results would require an

intermediate computation with many more than 56 bits (see Chapter 10 for a discussion of this topic).

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.

4.4.2 Wong and Goto's algorithm for computing exponentials

Assume that we want to compute e^x, where x is a normalized double-precision floating-point number:

$$x = s \times m \times 2^{exponent}.$$

The largest representable finite number in the IEEE-754 double-precision 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[9] $+\infty$.

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:

$$x \approx \hat{x} = s \times (p + 0.x_1x_2x_3\cdots x_{56}), 1 \leq p < 2^{10}.$$

If $|x| \geq 1/8$, this is done without any loss of accuracy, and if $|x| < 1/8$, we have

$$|e^x - e^{\hat{x}}| \leq 2^{-57}e^{1/8} \leq 1.14 \times 2^{-57},$$

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

$$e^x = e^p \times f_1 \times f_2 \times \cdots \times f_n, \tag{4.4}$$

where $f_i = e^{\alpha_i}$.

Unfortunately, this cannot be done without modifications, since (4.4) would require full double-precision 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

[9] 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 $+\infty$.

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:

$$f_1 = [e^{\alpha_1}]_{(-1)-9}\,.$$

Using a rectangular multiplier, we can compute $K_1 = e^p \times f_1$. 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:

   $$x = s \times (p + 0.x_1x_2x_3\cdots x_{56})\,, 1 \le p < 2^{10}.$$

   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.

   - $f_0 = e^p$ if $s = +1$, or e^{-p} 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:

   $$\begin{aligned} e^x \approx\ & f_0 \times f_1 \times f_2 \\ & \times \left[1 + r_2 + \frac{[r_2]_{17-24}}{2}\left([r_2]_{25-32} + [r_2]_{33-40}\right)\right. \\ & \left. + \frac{[r_2]^2_{17-24}}{2} + \frac{[r_2]^2_{25-32}}{2} + \frac{[r_2]^3_{17-24}}{6}\right]. \end{aligned}$$

4.4.3 Ercegovac et al.'s algorithm

Ercegovac, Lang, Muller and Tisserand [126] suggest to first perform a range reduction so that the input argument A becomes less than 2^{-k} for some 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 assuming that the Taylor expansion of f is

$$f(A) = C_0 + C_1 A + C_2 A^2 + C_3 A^3 + \cdots, \tag{4.5}$$

they use the following formula:

$$f(A) \approx C_0 A + C_1 A + C_2 A_2^2 z^4 + 2C_2 A_2 A_3 z^5 + C_3 A_2^3 z^6. \tag{4.6}$$

That formula is obtained by expanding the series (4.5) 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

$$\sqrt{1+A} = 1 + \frac{A}{2} - \frac{1}{8}A_2^2 z^4 - \frac{1}{4}A_2 A_3 z^5 + \frac{1}{16}A_2^3 z^6,$$

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 (4.6), 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 4.2 presents the "evaluation module" that computes $f(A)$.

4.4.4 Bipartite and multipartite methods

In [296], 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 representation of x into three 4-bit words, and to approximate the sine of $x = A + B + C$, where $A < \pi/2$, $B < 2^{-4}\pi/2$ and $C < 2^{-8}\pi/2$, using

$$\sin(A + B + C) \approx \sin(A + B) + \cos(A)\sin(C). \tag{4.7}$$

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*.

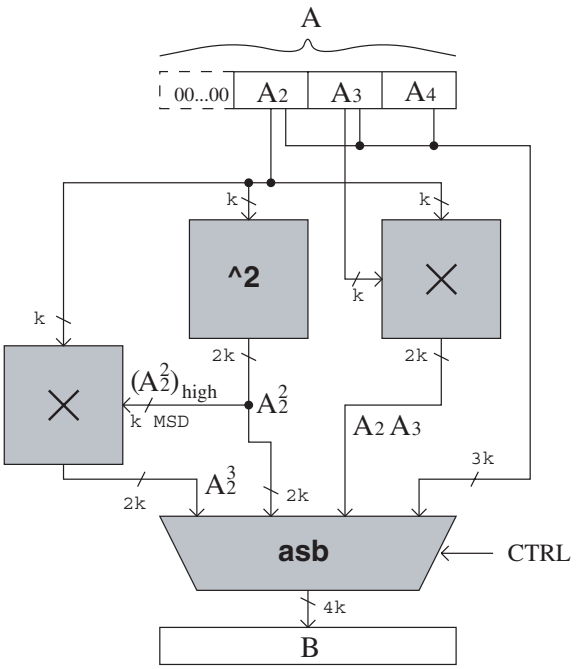


Figure 4.2: *The computation of $f(A)$ using Ercegovac et al.'s algorithm.*

That method was re-discovered (and named bipartite) by DasSarma and Matula [275]. 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 an n-bit fixed-point format and belong to $[\alpha, 1)$, where $\alpha > 0$ is some real number. Also, assume $n = 3k$. A straightforward tabulation of function f would require a table with n address bits. The size of that table would be $n \times 2^n$ bits. The first idea behind the bipartite method consists in splitting the input n-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 $0 \leq x_i < 1$. The 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 $|\epsilon_1| \leq 2^{-4k-1} \max_{[\alpha,1]} |f''|$. 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 $|\epsilon_2| \leq 2^{-k} \max_{[\alpha,1]} |f''|$. This gives the *bipartite formula*:

$$f(x) = A(x_0, x_1) + B(x_0, x_2) + \epsilon$$

where

$$\begin{aligned} A(x_0, x_1) &= f(x_0 + 2^{-k} x_1) \\ B(x_0, x_2) &= 2^{-2k} x_2 . f'(x_0) \\ \epsilon &\leq \left(2^{-4k-1} + 2^{-3k}\right) \max_{[\alpha,1]} |f''|. \end{aligned} \tag{4.8}$$

If the second derivative of f remains in the order of magnitude of 1, 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 in tabulating A and B instead of tabulating f. Since A and B are functions of $2k = 2n/3$ bits only, if A is rounded to the nearest n-bit number, if $2^{2k}B$ is rounded to the nearest $k = n/3$-bit number, and if the values of A and B are less than 1, then the total table size is

$$\left(\frac{4n}{3}\right) 2^{2n/3} \text{ bits}$$

which is a very significant improvement. Here, we implicitly assume that f and f'' have orders of magnitude that are close together. In other cases, the method must be adapted. For $n = 15$, 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

$$\left(2^{-4k-1} + 2^{-3k}\right) \max_{[\alpha,1]} |f''| + 2^{-3k}.$$

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,

$$\begin{cases} A(x_0, x_1) = 1/(x_0 + 2^{-k} x_1) \\ B(x_0, x_2) = -2^{-2k} x_2 / x_0^2 \\ \epsilon \leq 16 \left(2^{-4k-1} + 2^{-3k}\right). \end{cases}$$

To get better approximations, one can store A and B with some additional guard bits. Also, n 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 [279, 280] gave the general formula (4.8). They also suggested several improvements. The major one was to perform the first Taylor

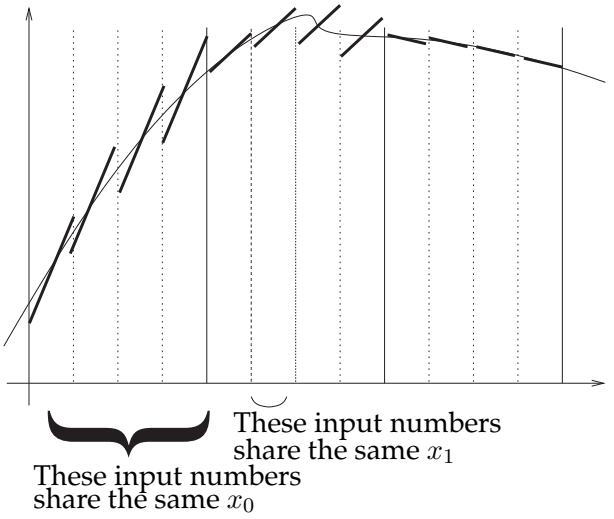


Figure 4.3: *The bipartite method is a piecewise linear approximation for which the slopes of the approximating straight lines are constants in intervals sharing the same value of* x_0.

expansion near[10] $x_0 + 2^{-k}x_1 + 2^{-2k-1}$ 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 4.3. Let us now give an example.

Example 7 (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 sub-words of different sizes:* x_0 *and* x_1 *are* 6*-bit numbers, and* x_2 *is a* 4*-bit*

[10] 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.

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

$$20 \times 2^{12} + 8 \times 2^{10} \textit{ bits } = 11 \textit{ kbytes}\,.$$

Obtaining a similar accuracy with a direct tabulation would require a table with

$$18 \times 2^{16} \textit{ bits } = 144 \textit{ kbytes}\,.$$

Schulte and Stine [283, 294] and Muller [238] 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 [97, 98] 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 uses bipartite ROM tables for the reciprocal and square root initial approximations [248].

4.4.5 Miscellaneous

Jain and Lin [178] use a combination of tabulation and "matched interpolation polynomials". Lee, Luk, Villasenor and Cheung [202, 203] suggest to use polynomial approximation with non-uniform segments. Lee, Mencer, Pearce and Wayne automate table-with-polynomial function evaluation for FPGAs [204]. Detrey and Dinechin [112] give a very efficient second order method, that requires one multiplication only. They have implemented it on FPGAs.

Chapter 5

Multiple-Precision Evaluation of Functions

5.1 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 double and extended-precision formats [15]. Some computations in "experimental mathematics" [33] are performed using hundreds or thousands of bits. For instance, multiple-precision calculations allowed Borwein, Plouffe and Bailey to find an algorithm for computing individual[1] hexadecimal digits of π. When we design algorithms for computing functions in single, double, extended, or quad-precision, 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. Also, the availability of an efficient and reliable multiple-precision library is of great help for testing floating-point software [44]. 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 [233, 266].

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 [15, 327].

In this domain, the pioneering work was done by Brent [40, 41] and Salamin [274]. Brent designed an arithmetic package named MP [43, 42]. It was

[1] That is, there is no need to compute any of the previous digits. For instance [33], the 2.5×10^{14}th hexadecimal digit of π is an E.

the first widely diffused package that was portable and contained routines for all common elementary and special functions. Bailey designed two packages, MPFUN [13] and ARPREC[2] [15]. In both packages, the high-level algorithms are similar, but ARPREC reaches high performance by a clever use of low-level algorithms similar to the Fast2Sum and Fast2Mult algorithms presented in Chapter 2 of this book. The PARI package[3], originally designed by Cohen and his research group, was especially designed for number theory and multiple-precision. The MPFR library[91] 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[153] multiple-precision library. MPFI is an extension of MPFR that implements multiple-precision interval arithmetic [266].

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 5.5) are 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 (double, extended or quad) 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.

5.2 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[4], but the performance of a multiple-precision system depends much on the performance of the multiplication programs: all other operations 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.

[2] See `http://crd.lbl.gov/~dhbailey/mpdist/`.

[3] See `http://pari.math.u-bordeaux.fr/`.

[4] In fact, it is an elementary function. The functions we deal with in this book should be called *elementary transcendental functions*.

5.2.1 Karatsuba's method

Assume we want to multiply two n-bit integers[5]

$$A = a_{n-1}a_{n-2}a_{n-3}\cdots a_0$$

and

$$B = b_{n-1}b_{n-2}b_{n-3}\cdots b_0,$$

and assume that n is even. We want to compute AB using $n/2$-bit $\times n/2$-bit multiplications. Define the following $n/2$-bit numbers:

$$\begin{cases} A^{(1)} = a_{n-1}a_{n-2}a_{n-3}\cdots a_{n/2} \\ A^{(0)} = a_{n/2-1}a_{n/2-2}a_{n/2-3}\cdots a_0 \\ B^{(1)} = b_{n-1}b_{n-2}b_{n-3}\cdots b_{n/2} \\ B^{(0)} = b_{n/2-1}b_{n/2-2}b_{n/2-3}\cdots b_0. \end{cases}$$

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,

$$AB = 2^n A^{(1)}B^{(1)} + 2^{n/2}(A^{(1)}B^{(0)} + A^{(0)}B^{(1)}) + A^{(0)}B^{(0)}. \tag{5.1}$$

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.

In 1962, Karatsuba and Ofman [184] noticed that[6]

$$\begin{aligned} AB = {} & 2^n A^{(1)}B^{(1)} \\ & + 2^{n/2}\left((A^{(1)} - A^{(0)})(B^{(0)} - B^{(1)}) + A^{(0)}B^{(0)} + A^{(1)}B^{(1)}\right) \\ & + A^{(0)}B^{(0)}. \end{aligned} \tag{5.2}$$

Using (5.2), 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

$$n^{\frac{\ln(3)}{\ln(2)}} \approx n^{1.585}.$$

[5] I give this presentation assuming radix 2 is used. Generalization to other radices is straightforward.

[6] This is not exactly Katatsuba and Ofman's presentation. We give here Knuth's version of the algorithm [187], which is slightly better.

Similar (and asymptotically more efficient) decompositions of a product can be found in the literature [187, 312, 330]. Montgomery [232] recently presented several Karatsuba-like formulae.

5.2.2 FFT-based methods

A. Schönhage and V. Strassen [278] noticed that Cooley and Tukey's FFT algorithm [79] can be used to perform fast multiplication of huge numbers. Their method allows one to multiply two n-bit numbers in time $O(n \ln n \ln \ln n)$.

The FFT-based multiplication requires significantly more memory than Karatsuba-like methods. As noted by Bailey and Borwein [33], 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 call $M(n)$ the delay of n-bit multiplication.

5.3 Multiple-Precision Division and Square-Root

5.3.1 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 [243], to solve polynomial equations (without an explicit use of the derivative), and generalized by Raphson a few years later [284]. NR-based division and/or square-root have been implemented on many recent processors [224, 226, 248, 264, 272].

Assume we want to compute a root α of some function φ. The NR iteration consists in building a sequence

$$x_{n+1} = x_n - \frac{\varphi(x_n)}{\varphi'(x_n)}. \tag{5.3}$$

If φ has a continuous derivative and if α is a single root (that is, $\varphi'(\alpha) \neq 0$), 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 [90]. 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 [145].

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.

- 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, usually a fairly "expensive" operation, 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[7], the complexity of square root evaluation or division

[7] 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 [41]. This assumption is satisfied by the grammar school multiplication method, as well as by the Karatsuba-like and the FFT-based algorithms.

is the same as that of multiplication: the time required to get $n = 2^k$ bits of a quotient is

$$\begin{aligned} &O\left(M(1) + M(2) + M(4) + M(8) + \cdots + M\left(2^{k-1}\right) + M\left(2^k\right)\right) \\ &\quad = O\left(M\left(2^k\right)\right) = O(M(n)). \end{aligned}$$

5.4 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 [13], 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**

  $$\exp(x) = \left(1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \frac{r^4}{4!} + \cdots\right)^{256} \times 2^n = \exp(r)^{256} \times 2^n,$$

  where elevating $\exp(r)$ to the power 256 only requires 8 consecutive squarings, since

  $$a^{256} = \left(\left(\left(\left(\left(\left(\left(a^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2;$$

whereas the logarithm of a is computed using the previous algorithm and the Newton iteration (5.3) with $\varphi(x) = \exp(x) - a$:

$$x_{n+1} = x_n + \frac{a - \exp(x_n)}{\exp(x_n)}.$$

This gives

$$x_n \to \ln(a).$$

5.5 The Arithmetic-Geometric (AGM) Mean

5.5.1 Presentation of the AGM

When a very high precision is required[8], Taylor series are no longer of interest, and we must use the *arithmetic-geometric mean* [34], 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:

$$\begin{cases} a_{n+1} = \dfrac{a_n + b_n}{2} \\ b_{n+1} = \sqrt{a_n b_n}. \end{cases}$$

$A(a_0, b_0)$ is called the *arithmetic-geometric mean* (AGM for short) of a_0 and b_0. Gauss noticed that $A(1, x)$ is equal to

$$\frac{\pi}{2F(x)},$$

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.

5.5.2 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

$$\begin{aligned} F(4/s) = \ln(s) + \frac{4\ln(s) - 4}{s^2} + \frac{36\ln(s) - 42}{s^4} \\ + \frac{1200\ln(s) - 1480}{3s^6} + O\left(\frac{1}{s^8}\right), \end{aligned} \tag{5.4}$$

[8] According to Borwein and Borwein [35], the switchover is somewhere in the 100 to 1000 decimal digit range.

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 (5.4) 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

$$\ln(x) \approx \frac{\pi}{2A(1, 4/s)} - m\ln(2). \tag{5.5}$$

Example: computation of $\ln(25)$.

Assume 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

$$\begin{cases} m = 496 \\ s \ = 25 \times 2^{96}. \end{cases}$$

We therefore have to compute $A(1, 4/s)$ using the AGM iteration, with

$$\begin{aligned} 4/s = 7.&82063709055898798605306724662686131146087101586515625956765160776010656390328437171675452187651898433760\\ &03908955915959176137047751789669625219407723543497974\\ &54654516566458501688411107378001622274090262189901259\\ &48905897827823885444189434604211742729022741015352086\\ &38673974261798262712603991118844289965640614870\cdots\\ &\times 10^{-151} \end{aligned}$$

After 16 iterations, we get

$$\begin{aligned} A(1, 4/s) = 0.&00452653127147250096072983021051330291298273470158382817131216067448690439957375794792338578019057417700371454676009364074131526473035830857845058014797497352119203403501958066084502701791872357638470143804008606271554404074490685737500078687218841447201069157798369164678109017756105713418996572318763110465033 9\cdots \end{aligned}$$

Hence,

$$\ln(25) \approx \frac{\pi}{2A(1,4/s)} - m\ln(2)$$

$$\begin{aligned}\approx\ & 3.21887582486820074920151866645237527905120270853703544382529578294835797541531552926026775618635922159993260604343112579944801045864935239926723323492\\ & 74114551043592749943664913069857124046830501145403103872017595547794513763870814255323094624436190\\ & 55897042585642716119445135344570574480923178896356782251142\cdots\end{aligned}$$

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 (5.5) 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 (5.4): 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 (5.4) and the AGM iteration. This gives

$$\ln(2) \approx \frac{\pi}{pA(1,2^{-p/2+2})}. \tag{5.6}$$

There are several ways of computing π [14]. We can for instance use the following algorithm, due to Brent [41] and Salamin [274], 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 5.1, converges quadratically to π.

k	p_k
1	3.18767264271210862720192997052536923265105357185936922648763
2	3.14168029329765329391807042456000938279571943881540283264419
3	3.14159265389544649600291475881804348610887923726131158965110
4	3.14159265358979323846636060270663132175770241134242935648685
5	3.14159265358979323846264338327950288419716994916472660583470
6	3.14159265358979323846264338327950288419716939937510582097494

Table 5.1: *The first terms of the sequence p_k generated by the Brent–Salamin algorithm. That sequence converges to π quadratically.*

n	x_n
1	$2.71828181384870854839394204546332554936588598573150616522\cdots$
2	$2.71828182845904519609615815000766898668384995737383488643\cdots$
3	$2.71828182845904523536028747135266221418255751906243823415\cdots$
4	$2.71828182845904523536028747135266249775724709369995957496\cdots$
5	$2.71828182845904523536028747135266249775724709369995957496\cdots$

Table 5.2: *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.*

5.5.3 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 (5.3) with $\varphi(x) = \ln(x) - a$. This gives

$$x_{n+1} = x_n \left(1 + a - \ln(x_n)\right), \tag{5.7}$$

where $\ln(x_n)$ is evaluated using the method given in Section 5.5.2. An example (computation of $\exp(1)$) is given in Table 5.2. 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.

5.5.4 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: (5.4) remains true if s is a complex number). Several variants have been suggested for these functions. Here is Brent's 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:

$$\begin{cases} q_{i+1} = 2q_i/(1+s_i) \\ a_i \quad = 2s_i v_i/(1+v_i^2) \\ b_i \quad = a_i/(1+\sqrt{1-a_i^2}) \\ c_i \quad = (v_i+b_i)/(1-v_i b_i) \\ v_{i+1} = c_i/(1+\sqrt{1+c_i^2}) \\ s_{i+1} = 2\sqrt{s_i}/(1+s_i) \end{cases}$$

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 $O(M(n)\ln n)$ [41]. 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 (5.3) becomes

$$x_{n+1} = x_n - (1+x_n^2)(\arctan(x_n) - \theta). \tag{5.8}$$

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 $O(M(n)\ln n)$ [41], i.e., with the same complexity as the arctangent function. As suggested by Brent,

function	complexity
addition	$O(n)$
multiplication	$O(n\ln(n)\ln\ln(n)))$
division, sqrt	$O(M(n))$
$\ln, \exp$	$O(M(n)\ln(n))$
$\sin, \cos, \arctan$	$O(M(n)\ln(n))$

Table 5.3: *Time complexity of the evaluation of some functions in multiple-precision arithmetic (extracted from Table 1 of [35]). $M(n)$ is the complexity of n-bit multiplication.*

$\sin(\theta)$ and $\tan(\theta)$ can then be computed using

$$\begin{aligned} x &= \tan\left(\frac{\theta}{2}\right) \\ \sin(\theta) &= \frac{2x}{1+x^2} \\ \cos(\theta) &= \frac{1-x^2}{1+x^2}. \end{aligned}$$

There are similar algorithms for many usual functions. Table 5.3 gives the time complexity for the evaluation of the most common ones.

Chapter 6

Introduction to Shift-and-Add Algorithms

At the beginning of the 17th 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 6.1 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

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

Table 6.1: *The filing of the different weights.*

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:

$$\begin{aligned}&\left(1+\frac{x_1}{10}\right)\left(1+\frac{x_2}{100}\right)\left(1+\frac{x_3}{1,000}\right)\\ &\quad\simeq 1+\frac{x_1}{10}+\frac{x_2}{100}+\frac{x_3}{1,000}+\frac{x_1x_2}{1,000}+\frac{x_1x_3}{10,000}.\end{aligned}$$

Although this story is pure fiction (it was invented by Xavier Merrheim to defend his Ph.D. dissertation [229]), 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 [276]. He published 15-digit accurate tables in his *Arithmetica Logarithmica* (1624).

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.

6.1 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 6 (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

$$\begin{aligned} t_{n+1} &= t_n + d_n \ln\left(1 + 2^{-n}\right) \\ E_{n+1} &= E_n\left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n} \\ d_n &= \begin{cases} 1 \textit{ if } t_n + \ln\left(1 + 2^{-n}\right) \leq t \\ 0 \textit{ otherwise.} \end{cases} \end{aligned} \tag{6.1}$$

This algorithm only requires additions and multiplications by powers of 2. Such multiplications reduce to shifts when implemented in radix-2 arithmetic. Figures 6.1 through 6.3 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 towards a regular function of t as n goes towards 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

$$\lim_{n \to +\infty} t_n = t. \tag{6.2}$$

Since t_n is obviously equal to $\sum_{i=0}^{n-1} d_n \ln\left(1 + 2^{-n}\right)$, this implies

$$t = \sum_{i=0}^{\infty} d_n \ln\left(1 + 2^{-n}\right). \tag{6.3}$$

Thus it seems that our algorithm makes it possible to decompose any number t belonging to I into a sum

$$t = d_0 w_0 + d_1 w_1 + d_2 w_2 + \cdots,$$

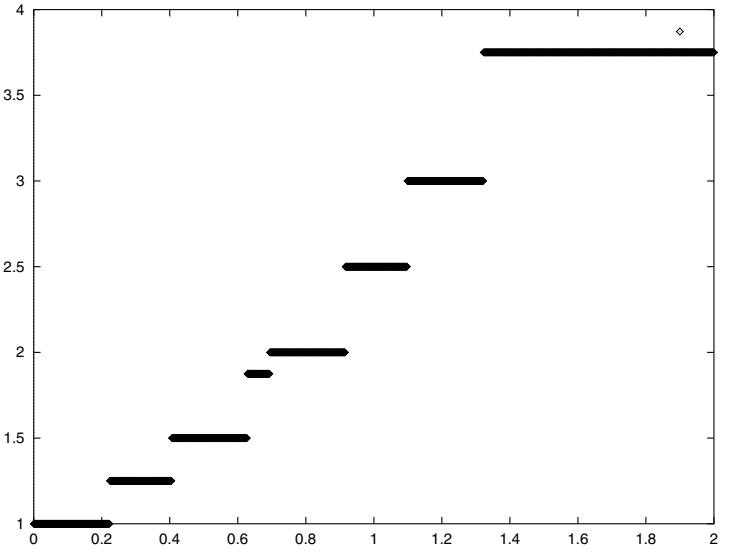


Figure 6.1: *Value of E_3 versus t.*

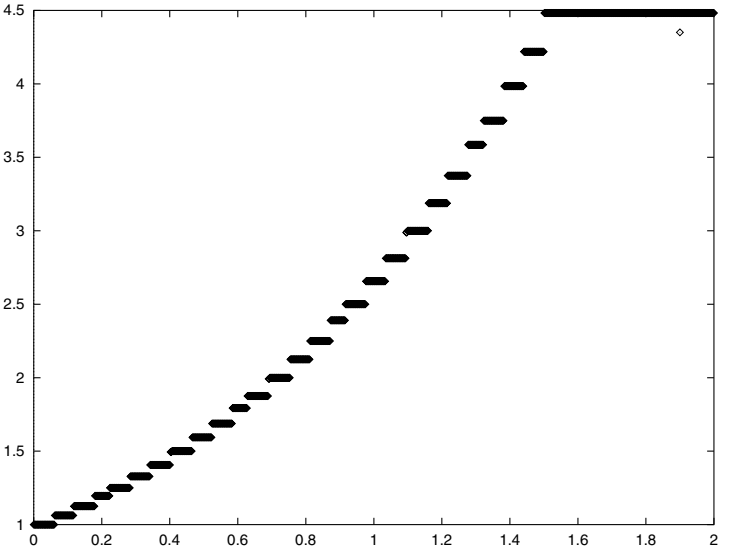


Figure 6.2: *Value of E_5 versus t.*

where $w_i = \ln(1 + 2^{-i})$. 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.

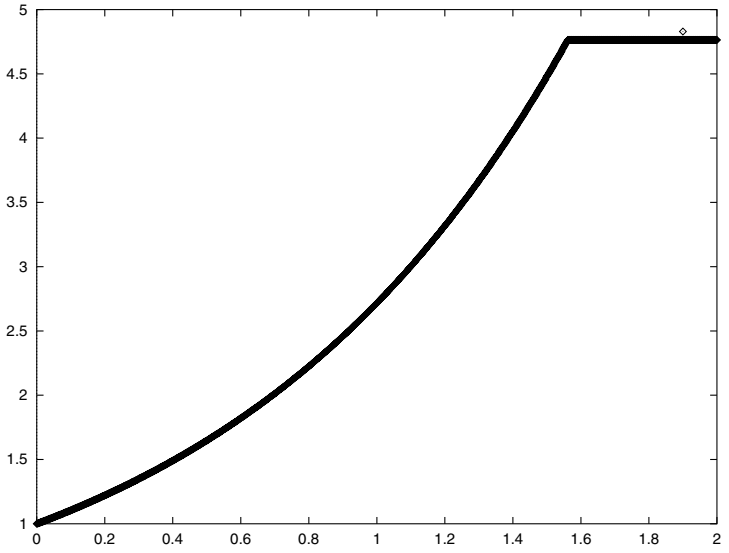


Figure 6.3: *Value of E_{11} versus t.*

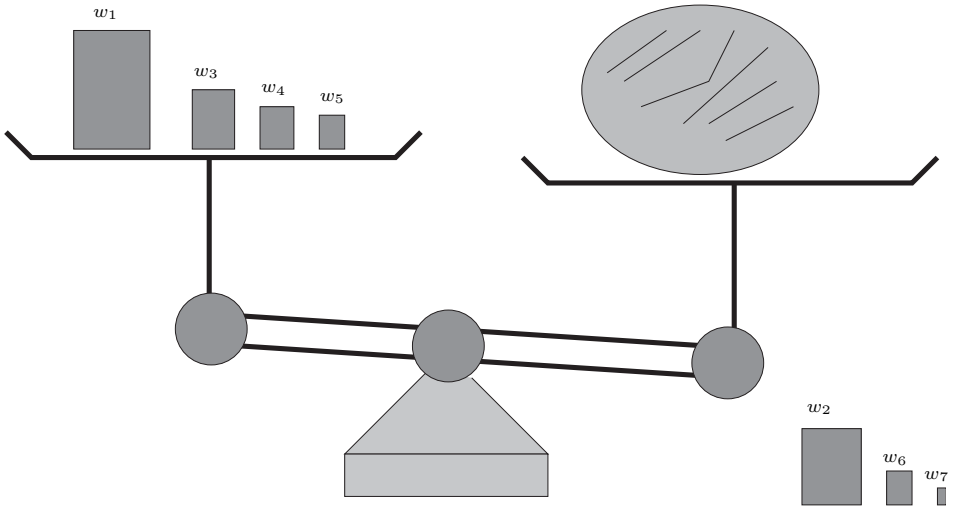


Figure 6.4: *The restoring algorithm. The weights are either unused or put on the pan that does not contain the loaf of bread being weighed. In this example, the weight of the loaf of bread is $w_1 + w_3 + w_4 + w_5 + \cdots$.*

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 6.4 illustrates this.

Theorem 11 (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*

$$\forall n, w_n \leq \sum_{k=n+1}^{\infty} w_k, \tag{6.4}$$

then for any $t \in [0, \sum_{k=0}^{\infty} w_k]$, the sequences (t_n) and (d_n) defined as

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } t_n + w_n \leq t \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{6.5}$$

satisfy

$$t = \sum_{n=0}^{\infty} d_n w_n = \lim_{n \to \infty} t_n.$$

Proof
Let us prove by induction that for any n,

$$0 \leq t - t_n \leq \sum_{k=n}^{\infty} w_k. \tag{6.6}$$

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 (6.6) would suffice to prove the theorem. Relation (6.6) 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 (6.4), 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 11 is called a *discrete base* [234, 235, 236].

The algorithm given by (6.5) is called here the "restoring algorithm", by analogy with the restoring division algorithm [135] (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 $t/y = d_0.d_1d_2d_3\cdots$). To our knowledge, this analogy between some division algorithms and most shift-and-add elementary function algorithms was first pointed out by Meggitt [228], who presented algorithms similar to those in this chapter as "pseudomultiplication" and "pseudodivision" methods. Other "pseudodivision" algorithms were suggested by Sarkar and

Krishnamurthy [273]. An algorithm very similar to the restoring exponential algorithm was suggested by Chen [58]. 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 7).

Theorem 12 (nonrestoring decomposition algorithm) *Let (w_n) be a decreasing sequence of positive real numbers such that $\sum_{i=0}^{\infty} w_i$ converges. If*

$$\forall n, w_n \leq \sum_{k=n+1}^{\infty} w_k, \tag{6.7}$$

then for any $t \in [-\sum_{k=0}^{\infty} w_k, \sum_{k=0}^{\infty} w_k]$, the sequences (t_n) and (d_n) defined as

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } t_n \leq t \\ -1 & \text{otherwise} \end{cases} \end{aligned} \tag{6.8}$$

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 11. 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 6.5 illustrates this.

6.2 Simple Algorithms for Exponentials and Logarithms

6.2.1 The restoring algorithm for exponentials

We admit that the sequences $\ln(1 + 2^{-n})$ and $\arctan 2^{-n}$ are discrete bases, that is, that they satisfy the conditions of Theorems 11 and 12 (see [234] for proof). Now let us again find Algorithm 6.1 using Theorem 11. We use the discrete base $w_n = \ln(1 + 2^{-n})$. Let $t \in [0, \sum_{k=0}^{\infty} w_k] \approx [0, 1.56202\cdots]$. From Theorem 11, the sequences (t_n) and (d_n) defined as

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n \ln(1 + 2^{-n}) \\ d_n &= \begin{cases} 1 & \text{if } t_n + \ln(1 + 2^{-n}) \leq t \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

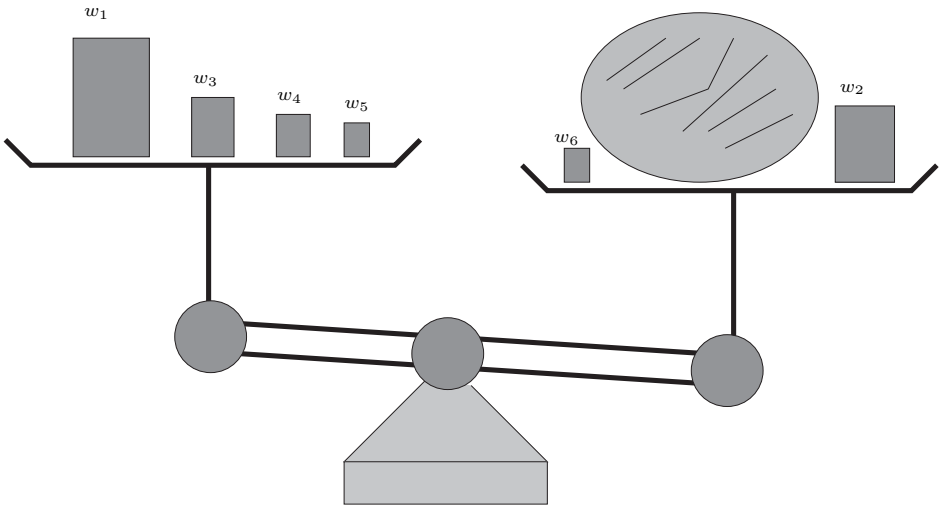


Figure 6.5: *The nonrestoring algorithm. All the weights are used, and they can be put on both pans. In this example, the weight of the loaf of bread is* $w_1 - w_2 + w_3 + w_4 + w_5 - w_6 + \cdots$.

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,

$$E_n = \exp\left(t_n\right). \tag{6.9}$$

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 (6.9) 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 6.1 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

$$\left[0, \sum_{n=0}^{\infty} \log_a\left(1 + 2^{-n}\right)\right].$$

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 11,

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 $E_n = \exp(t_n)$ we deduce:

$$1 \leq \exp(t - t_n) \leq \exp\left(2^{-n+1}\right).$$

This gives

$$\left|\frac{e^t - E_n}{e^t}\right| \leq 1 - e^{-2^{-n+1}} \leq 2^{-n+1}. \tag{6.10}$$

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).

6.2.2 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:

$$\begin{aligned} t_0 &= 0 \\ E_1 &= 1 \\ t_{n+1} &= t_n + d_n \ln\left(1 + 2^{-n}\right) \\ E_{n+1} &= E_n + d_n E_n 2^{-n} \end{aligned} \tag{6.11}$$

with

$$d_n = \begin{cases} 1 & \text{if } t_n + \ln\left(1 + 2^{-n}\right) \leq \ell \\ 0 & \text{otherwise.} \end{cases} \tag{6.12}$$

The previous study shows that, using this algorithm:

$$\begin{aligned} \lim_{n\to\infty} t_n &= \ell \\ \lim_{n\to\infty} E_n &= \exp(\ell) = x. \end{aligned}$$

[1] In this estimation, we do not take the rounding errors into account.

Of course, this algorithm cannot be used to compute ℓ since (6.12) 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 (6.12) is equivalent to the test:

$$d_n = \begin{cases} 1 & \text{if } E_n \times (1 + 2^{-n}) \leq x \\ 0 & \text{otherwise.} \end{cases} \tag{6.13}$$

Consequently, if we replace (6.12) by (6.13) 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.*

Algorithm 7 (restoring logarithm)

- **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

$$\begin{aligned} t_0 &= 0 \\ E_0 &= 1. \end{aligned}$$

Build two sequences t_i and E_i as follows

$$\begin{aligned} t_{i+1} &= t_i + \ln\left(1 + d_i 2^{-i}\right) \\ E_{i+1} &= E_i\left(1 + d_i 2^{-i}\right) = E_i + d_i E_i 2^{-i} \\ d_i &= \begin{cases} 1 & \textit{if } E_i + E_i 2^{-i} \leq x \\ 0 & \textit{otherwise.} \end{cases} \end{aligned} \tag{6.14}$$

As in the previous section, if we replace, in the algorithm, the constants $\ln(1 + 2^{-n})$ by $\log_a(1 + 2^{-n})$, we get an algorithm that evaluates base-a logarithms.

Error evaluation

From the proof of Theorem 11, 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}.

[2] In this estimation, we do not take the rounding errors into account.

6.3 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 [135, 270, 271], using redundant number systems. The algorithms presented in this section are slight variations of algorithms proposed by N. Takagi in his Ph.D. dissertation [300].

6.3.1 Faster computation of exponentials

First, we try to compute exponentials in a more efficient way. Let us start from the basic iteration (6.1). 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:

$$\begin{aligned} L_{n+1} &= L_n - \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 2^{-n}\right) \\ d_n &= \begin{cases} 1 & \text{if } L_n \geq \ln\left(1 + 2^{-n}\right) \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{6.15}$$

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

[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 if we were able to implement such huge tables efficiently, we would simply tabulate the exponential function.

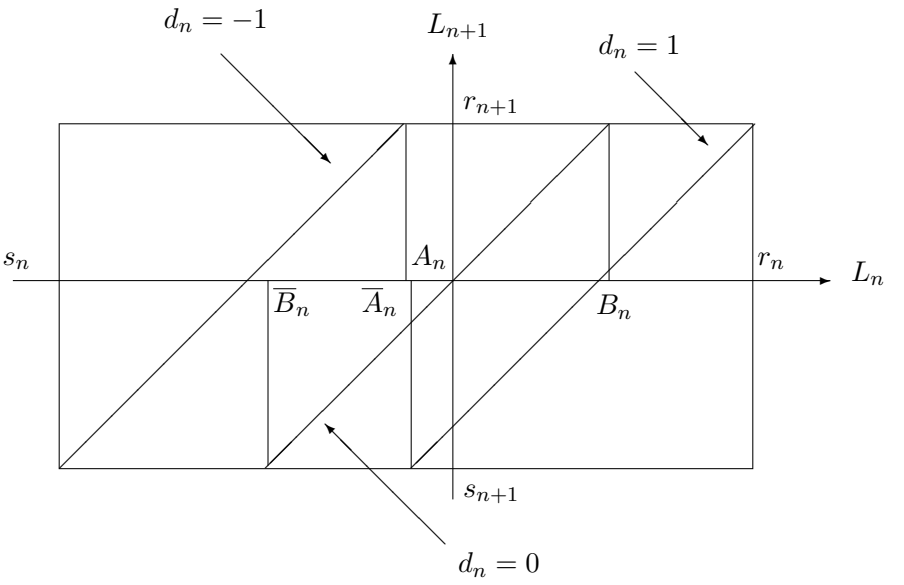


Figure 6.6: *Robertson diagram of the "redundant exponential" algorithm.*

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

$$\begin{aligned} r_n &= \sum_{k=n}^{\infty} \ln\left(1 + 2^{-k}\right) \\ s_n &= \sum_{k=n}^{\infty} \ln\left(1 - 2^{-k}\right). \end{aligned} \tag{6.16}$$

Figure 6.6 presents the different possible values of L_{n+1} versus L_n (following (6.15)) depending on the choice of d_n. This figure is very close to the *Robertson diagrams* that appear in the SRT division algorithm [135, 270], so we call it the *Robertson diagram* of iteration (6.15).

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 $L_n \geq A_n$.

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

Table 6.2: *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* $2^n \overline{B}_n$.

The values A_n, B_n, $\overline{A}_n$, and $\overline{B}_n$ of Figure 6.6 satisfy:

$$\begin{cases} A_n = s_{n+1} + \ln\left(1 + 2^{-n}\right) \\ B_n = r_{n+1} \\ \overline{A}_n = r_{n+1} + \ln\left(1 - 2^{-n}\right) \\ \overline{B}_n = s_{n+1}. \end{cases} \tag{6.17}$$

Using these relations and the Taylor expansion of the function $\ln(1+x)$, one can show that:

$$\begin{cases} 2^n A_n \leq & 0 \\ 2^n B_n \geq & 1/2 \\ 2^n \overline{A}_n \geq & -1/2 \\ 2^n \overline{B}_n \leq & -1. \end{cases} \tag{6.18}$$

Table 6.2 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

$$|L_n^* - 2^n L_n| \leq 1/2.$$

Therefore if we choose[4]

$$d_n = \begin{cases} -1 & \text{if } L_n^* \leq -1 \\ 0 & \text{if } -1/2 \leq L_n^* \leq 0 \\ 1 & \text{if } L_n^* \geq 1/2, \end{cases} \tag{6.19}$$

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 6.6) 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:

$$\begin{cases} -5/2 < 2^n s_n \\ 2^n r_n < 2; \end{cases} \tag{6.20}$$

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 $1/2$,

$$-\frac{5}{2} \leq L_n^* \leq 2.$$

[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 "$L_n^* \in \{-1/2, 0\}$."

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

$$L_n^* = \cdots L_{n,4}^* L_{n,3}^* L_{n,2}^* L_{n,1}^* L_{n,0}^* . L_{n,-1}^*,$$

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.[6]
- 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.

[6] Thus, if the selection of d_n is implemented by a PLA, any value that minimizes the size of the PLA can be chosen.

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 [135]. A slightly different solution, used by Takagi [300], 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 7.5.

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

$$d_n = \begin{cases} -1 & \text{if } L^*_n \leq -3/2 \\ 0 & \text{if } -1 \leq L^*_n \leq -1/2 \\ 1 & \text{if } L^*_n \geq 0, \end{cases} \tag{6.21}$$

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.

6.3.2 Faster computation of logarithms

Now assume that we want to compute logarithms quickly. Some notations adopted here are taken from [239], and Asger-Munk Nielsen helped to perform this study. We start from (6.14), 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,

$$\begin{cases} L_{n+1} = L_n - \ln\left(1 + d_n 2^{-n}\right) \\ E_{n+1} = E_n \left(1 + d_n 2^{-n}\right) = E_n + d_n E_n 2^{-n}, \end{cases} \tag{6.22}$$

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 $L_n \to L_1 + \ln(E_1)$. 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 6.7 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 $r_{n+1} = (1 - 2^{-n})\, r_n - 2^{-n}$, and s_n satisfies $s_{n+1} = (1 + 2^{-n})\, s_n + 2^{-n}$. This gives:

$$\begin{aligned} r_n &= \sum_{k=n}^{\infty} 2^{-k} \prod_{j=n}^{k} \frac{1}{1 - 2^{-j}} \\ s_n &= -\sum_{k=n}^{\infty} 2^{-k} \prod_{j=n}^{k} \frac{1}{1 + 2^{-j}}. \end{aligned}$$

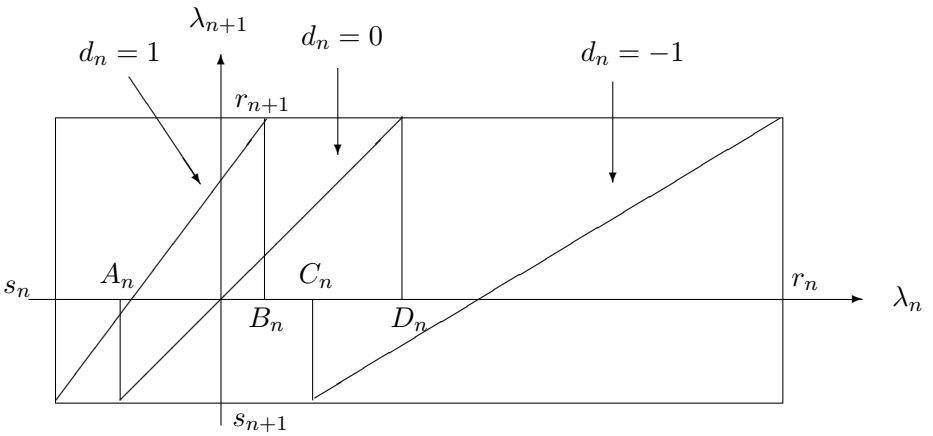


Figure 6.7: *Robertson diagram for the logarithm. The three straight lines give* $\lambda_{n+1} = \lambda_n(1 + d_n 2^{-n}) + d_n 2^{-n}$ *for* $d_n = -1, 0, 1$.

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 (6.23) will ensure $\lambda_n \in [s_n, r_n]$, which implies $\lambda_n \to 0$.

$$\begin{cases} \text{if } \lambda_n < A_n & \text{then } d_n = 1 \\ \text{if } A_n \le \lambda_n \le B_n & \text{then } d_n = 1 \text{ or } 0 \\ \text{if } B_n < \lambda_n < C_n & \text{then } d_n = 0 \\ \text{if } C_n \le \lambda_n \le D_n & \text{then } d_n = 0 \text{ or } -1 \\ \text{if } D_n < \lambda_n & \text{then } d_n = -1. \end{cases} \tag{6.23}$$

The values A_n, B_n, C_n, and D_n satisfy

$$A_n = s_{n+1}$$

$$B_n = \frac{r_{n+1} - 2^{-n}}{1 + 2^{-n}}$$

$$C_n = \frac{s_{n+1} + 2^{-n}}{1 - 2^{-n}}$$

$$D_n = r_{n+1}.$$

It follows, using these relations, that $A_n < B_n < C_n < D_n$ for all $n \ge 1$. Table 6.3 gives the first values and limits of 2^n times these values.

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

Table 6.3: *First values and limits of* $2^n s_n$, $2^n r_n$, $2^n A_n$, $2^n B_n$, $2^n C_n$, *and* $2^n D_n$.

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

$$|\lambda_n^* - 2^n \lambda_n| \leq \frac{1}{2}.$$

Therefore, *if* n *is greater than or equal to* 2, we can choose

$$d_n = \begin{cases} +1 & \text{if } \lambda_n^* \leq -1/2 \\ 0 & \text{if } \lambda_n^* = 0 \text{ or } 1/2 \\ -1 & \text{if } \lambda_n^* \geq 1. \end{cases} \tag{6.24}$$

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 (6.24) 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

$$L_1 \in [s_1 + 1, r_1 + 1] = [0.4194\cdots, 3.4627\cdots].$$

6.4 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 6.4). Baker's *predictive algorithm* [22], originally designed for computing the trigonometric functions[7] 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

$$0 < 2^{-i} - \ln(1 + 2^{-i}) < 2^{-2i-1}. \tag{6.25}$$

Similarly, one can show that

$$0 < 2^{-i} - \arctan(2^{-i}) < \frac{1}{3}2^{-3i}. \tag{6.26}$$

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 6.5.

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:

$$0 \leq x_n \leq \sum_{k=n}^{\infty} w_k < \sum_{k=n}^{\infty} 2^{-k} = 2^{-n+1}$$

[7] Using a modified CORDIC algorithm; CORDIC is presented in the next chapter.

i	$\ln(1+2^{-i})$
0	$0.10110001011100100001011111110111101000111\cdots$
1	$0.01100111110011001000111110110010111111111001\cdots$
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	$0.00000000001111111111100000000001010101010 0\cdots$
11	$0.00000000000111111111111000000000010101010\cdots$
12	$0.00000000000011111111111110000000000010101\cdots$
13	$0.00000000000001111111111111100000000000010\cdots$
14	$0.00000000000000111111111111111000000000000 0\cdots$
15	$0.00000000000000011111111111111110000000000 0\cdots$

Table 6.4: *The first digits of the first 15 values* $w_i = \ln(1+2^{-i})$. *As* i *increases,* w_i *gets closer to* 2^{-i}.

x	w_i	decomposition
$\frac{1}{2}$	2^{-i} (binary)	01000000000000000000
	$\ln(1+2^{-i})$	01001100011001101 1101
	$\arctan 2^{-i}$	01000100101001111 0000
$\frac{1}{10}$	2^{-i} (binary)	00001100110011001 1001
	$\ln(1+2^{-i})$	00001101000110101 1101
	$\arctan 2^{-i}$	00001100110011111 1001
$\frac{1}{1024}$	2^{-i} (binary)	00000000001000000000000000000000000000
	$\ln(1+2^{-i})$	00000000001000000000001111111111010101
	$\arctan 2^{-i}$	00000000001000000000000000000000010101

Table 6.5: *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.*

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:

$$\begin{cases} d_n &= d_n^{(n)} \\ d_{n+1} &= d_{n+1}^{(n)} \\ &\vdots \\ d_\ell &= d_\ell^{(n)}, \end{cases} \tag{6.27}$$

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 (6.25), this gives

$$0 \le \tilde{x}_{\ell+1} < 2^{-\ell} + \frac{2^{-2n+1}}{3}, \tag{6.28}$$

- and if $w_i = \arctan 2^{-i}$, using (6.26), this gives

$$0 \le \tilde{x}_{\ell+1} < 2^{-\ell} + \frac{2^{-3n+3}}{21}. \tag{6.29}$$

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

$$0 \leq \tilde{x}_{2n} \leq 2^{-2n+1}\left(1 + \frac{1}{3}\right).$$

  The "correction step" consists of again using the constant $w_\ell = w_{2n-1}$ in the decomposition.[8] 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 $0 \leq x_{\ell+1} \leq w_\ell \leq \sum_{k=l+1}^{\infty} w_k$;
  - if $\delta_\ell = 1$, then from (6.28) and the Taylor expansion of the logarithm, we get:

$$\begin{aligned} 0 \leq x_{\ell+1} = \tilde{x}_{\ell+1} - w_l &\leq 2^{-2n+1}\left(1 + \frac{1}{3}\right) - 2^{-2n+1} \\ &\quad + \frac{1}{2}2^{-4n+2} \\ &\leq 2^{-2n}\left(\frac{2}{3} + 2^{-2n+1}\right) \leq 2^{-2n} \\ &\quad \text{(since } n \geq 2\text{)} \\ &\leq \sum_{k=l+1}^{\infty} w_k. \end{aligned}$$

- 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

$$0 \leq x_{\ell+1} \leq w_\ell \leq \sum_{k=\ell+1}^{\infty} w_k;$$

[8] 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 $\delta_\ell = 1$, then, from (6.29) 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 $w_n = \ln(1 + 2^{-n})$. 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,[9] 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 $2 \le n \le 8$, 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 $2 \le n \le 10$, $m = n + 1$ 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

$$x_n = \begin{cases} x_n^{(0)} & \text{if } x_n^{(1)} < 0 \\ x_n^{(1)} & \text{otherwise.} \end{cases}$$

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;
# recalculation of the convergence domain [0,A]
```

[9] That is, $0 \le x_0 \le \sum_{k=0}^{\infty} w_k$.

```
    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 8 (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 6.6, 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,* $0.000010111110110001111 0\cdots_2$. *This gives*

$$d_4 = 0, d_5 = 1, d_6 = 0, d_7 = 1;$$

therefore

$$\begin{aligned} \tilde{x}_8 &= x_4 - d_4 w_4 - d_5 w_5 - d_6 w_6 - d_7 w_7 \\ &= 0.0080232558\cdots_{10} \\ &= 0.0000010000001101110\cdots_2. \end{aligned}$$

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*

$$\begin{aligned} x_8 &= \tilde{x}_8 - \delta_7 w_7 \\ &= 0.0002411153\cdots_{10} \\ &= 0.00000000000011111100110100111110101\cdots_2. \end{aligned}$$

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

Table 6.6: *Table obtained for $n = 4$ using our Maple program.*

From the binary representation of x_8, we can deduce $d_8, d_9, \ldots, d_{15}$. 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

$$\begin{aligned} x = {} & w_1 + w_2 + w_3 && \text{tabulation and correction} \\ & + w_5 + w_7 + w_7 && \text{1st step of Baker's method} \\ & + w_{13} + w_{14} + w_{15} + \cdots && \text{2nd step of Baker's method.} \end{aligned}$$

This gives

$$\begin{aligned} e^x = &\left(1+\frac{1}{2^{-1}}\right)\left(1+\frac{1}{2^{-2}}\right)\left(1+\frac{1}{2^{-3}}\right) \\ &\left(1+\frac{1}{2^{-5}}\right)\left(1+\frac{1}{2^{-7}}\right)\left(1+\frac{1}{2^{-7}}\right) \\ &\left(1+\frac{1}{2^{-13}}\right)\left(1+\frac{1}{2^{-14}}\right)\left(1+\frac{1}{2^{-15}}\right) \\ \approx &\ 2.2099. \end{aligned}$$

6.5 Bibliographic Notes

The CORDIC algorithm (see Chapter 7) is a shift-and-add algorithm that allows evaluation of trigonometric and hyperbolic functions. It was introduced by Volder in 1959 [317]. Meggitt [228] 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 [290] in 1965. Similar algorithms were studied by Linhardt and Miller [218]. An analysis of shift-and-add algorithms for computing the elementary functions was given by DeLugish [108] in 1970.

Chapter 7

The CORDIC Algorithm

7.1 Introduction

The CORDIC algorithm was introduced in 1959 by Volder [317, 318]. 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 [318], 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 [321, 322] 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 [63], and in arithmetic coprocessors such as the Intel 8087 [241]. Some authors have proposed the use of CORDIC processors for signal processing applications (DFT [26, 111, 329], discrete Hartley transform [56], filtering [109], SVD [53, 54, 134, 169, 200]), or for solving linear systems [3].

[1] See `http://www.decodesystems.com/hp9100.html`.

7.2 The Conventional CORDIC Iteration

Volder's algorithm is based upon the following iteration,

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n}. \end{cases} \tag{7.1}$$

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

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K \times \begin{pmatrix} x_0 \cos z_0 - y_0 \sin z_0 \\ x_0 \sin z_0 + y_0 \cos z_0 \\ 0 \end{pmatrix}, \tag{7.2}$$

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:

$$\begin{aligned} x_0 &= 1/K = 0.6072529350088812561694\cdots \\ y_0 &= 0 \\ z_0 &= \theta. \end{aligned}$$

Now let us show how CORDIC works. That algorithm is based on the decomposition of $\theta = z_0$ on the discrete base (see Chapter 6) $w_n = \arctan 2^{-n}$, using the nonrestoring algorithm (see Theorem 12). 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

$$\begin{aligned} t_0 &= 0 \\ t_{n+1} &= t_n + d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } t_n \leq \theta \\ -1 & \text{otherwise;} \end{cases} \end{aligned} \tag{7.3}$$

nth rotation

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} \cos(d_n w_n) & -\sin(d_n w_n) \\ \sin(d_n w_n) & \cos(d_n w_n) \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}. \tag{7.4}$$

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 by using the relation $\tan w_n = 2^{-n}$. We then replace (7.4) by:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \cos(w_n) \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}. \tag{7.5}$$

In (7.5), 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 (7.5), we perform:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix}, \tag{7.6}$$

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 (7.3) can be slightly simplified. If we define $z_n = \theta - t_n$, we get:

$$\begin{aligned} z_0 &= \theta \\ z_{n+1} &= z_n - d_n w_n \\ d_n &= \begin{cases} 1 & \text{if } z_n \geq 0 \\ -1 & \text{otherwise.} \end{cases} \end{aligned} \tag{7.7}$$

To sum up, the sequence (x_n, y_n) defined by Eq. (7.6) and (7.7) 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 (7.2). Figure 7.1 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 6.2.2). 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

$$\begin{cases} x_{n+1} = x_n - d_n y_n 2^{-n} \\ y_{n+1} = y_n + d_n x_n 2^{-n} \\ z_{n+1} = z_n - d_n \arctan 2^{-n} \end{cases}$$

[2]Or, more precisely, a similarity.

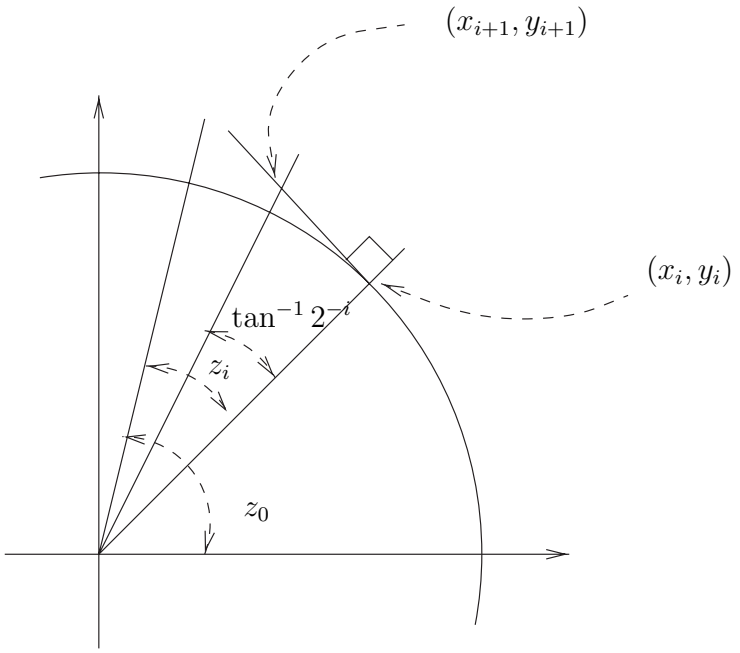


Figure 7.1: *One iteration of the CORDIC algorithm.*

with $z_0 = -\theta$ and

$$d_n = \begin{cases} 1 & \text{if } z_n \geq 0 \\ -1 & \text{otherwise.} \end{cases} \tag{7.8}$$

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 (7.8):

$$d_n = \begin{cases} 1 & \text{if } z'_n \geq \theta \\ -1 & \text{otherwise} \end{cases} \tag{7.9}$$

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

[3]Neglecting the scale factor.

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:

$$\lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = \begin{pmatrix} K\sqrt{x_0^2 + y_0^2} \\ 0 \\ z_0 + \arctan \dfrac{y_0}{x_0} \end{pmatrix}, \tag{7.10}$$

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 [321] found the correct modification, and obtained the generalized CORDIC iteration:

$$\begin{cases} x_{n+1} = x_n - m d_n y_n 2^{-\sigma(n)} \\ y_{n+1} = y_n + d_n x_n 2^{-\sigma(n)} \\ z_{n+1} = z_n - d_n w_{\sigma(n)}, \end{cases} \tag{7.11}$$

where the results and the values of d_n, m, w_n, and $\sigma(n)$ are presented in Tables 7.1 and 7.2.

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 Theorem 12. 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 $1.1181730155265\cdots$.

Type	m	w_k	$d_n = \text{sign} z_n$ (Rotation Mode)	$d_n = -\text{sign} y_n$ (Vectoring Mode)
circular	1	$\arctan 2^{-k}$	$x_n \to K\left(x_0 \cos z_0 - y_0 \sin z_0\right)$ $y_n \to K\left(y_0 \cos z_0 + x_0 \sin z_0\right)$ $z_n \to 0$	$x_n \to K\sqrt{x_0^2 + y_0^2}$ $y_n \to 0$ $z_n \to z_0 + \arctan \frac{y_0}{x_0}$
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	$\tanh^{-1} 2^{-k}$	$x_n \to K'\left(x_1 \cosh z_1 + y_1 \sinh z_1\right)$ $y_n \to K'\left(y_1 \cosh z_1 + x_1 \sinh z_1\right)$ $z_n \to 0$	$x_n \to K'\sqrt{x_1^2 - y_1^2}$ $y_n \to 0$ $z_n \to z_1 + \tanh^{-1} \frac{y_1}{x_1}$

Table 7.1: *Computability of different functions using CORDIC.*

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 $3^{k+1} + 2k - 1 \leq 2n$

Table 7.2: *Values of $\sigma(n)$ in Eq. (7.11) and Table 7.1.*

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

$$\sqrt{x} = K'\sqrt{\left(x + \frac{1}{4K'^2}\right)^2 - \left(x - \frac{1}{4K'^2}\right)^2}.$$

7.3 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

$$K = \prod_{i=0}^{+\infty} \sqrt{1 + 2^{-2i}}.$$

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 [111] and Haviland and Tuszinsky [162] 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:

$$\begin{aligned} x_{n+1} &= (x_n - d_n y_n 2^{-n})(1 + \alpha_n 2^{-n}) \\ y_{n+1} &= (y_n + d_n x_n 2^{-n})(1 + \alpha_n 2^{-n}). \end{aligned}$$

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

Table 7.3: *First values α_i that can be used in Despain's scale factor compensation method.*

There are several possible solutions for the sequence (α_n). One of them is given in Table 7.3.

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

> 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
```

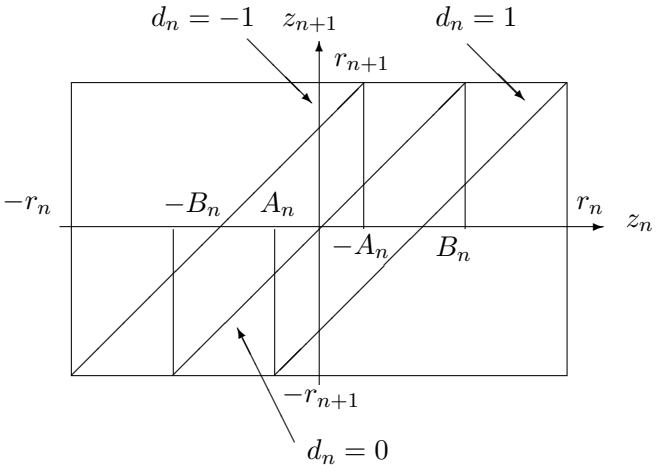


Figure 7.2: *Robertson diagram of CORDIC.*

Other scale factor compensation methods have been suggested. For instance, Deprettere, Dewilde, and Udo [109] choose, instead of rotation angles of the form $\arctan 2^{-n}$, angles of the form $\arctan(2^{-n} \pm 2^{-m})$, 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 [310].

7.4 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 7.2.

The values r_n, A_n, and B_n in Figure 7.2 satisfy:

$$\begin{cases} r_n = \displaystyle\sum_{k=n}^{\infty} \arctan 2^{-k} \\ B_n = r_{n+1} \\ A_n = -r_{n+1} + \arctan 2^{-n}. \end{cases}$$

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

Table 7.4: *First four values of* $2^n r_n$, $2^n A_n$ *and* $2^n B_n$.

Table 7.4 gives the first four values of $2^n r_n$, $2^n A_n$, and $2^n B_n$. One can easily show that:

$$A_n \leq 0$$

$$2^n B_n > 1/2.$$

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.

7.4.1 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

$$|z_n^* - 2^n z_n| \leq 1/2.$$

Therefore if we choose

$$d_n = \begin{cases} -1 & \text{if } z_n^* < 0 \\ 0 & \text{if } z_n^* = 0 \\ 1 & \text{if } z_n^* > 0, \end{cases} \tag{7.12}$$

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 7.2) 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 7.2) 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 $-r_{n+1} \leq z_{n+1} \leq r_{n+1}$.

7.4.2 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

$$d_n = \begin{cases} -1 & \text{if } z_n^* < -1/2 \\ 0 & \text{if } z_n^* = -1/2 \\ 1 & \text{if } z_n^* > -1/2, \end{cases} \tag{7.13}$$

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.

7.4.3 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, $1/K$) *on the fly*, in parallel with the CORDIC iterations. This was suggested by Ercegovac and Lang [131, 134];
- 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.

7.5 The Double Rotation Method

This method was suggested independently by Takagi et al. [300, 301, 302] and by Delosme [106, 167], 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 number system, the vectoring operation and the rotation by half the resulting angle. The basic idea behind the double rotation method, illustrated in Figure 7.3 is to perform the similarities of angle $d_i \arctan 2^{-i}$ *twice*. Assume that we are in the circular ($m = 1$) type of iterations, in rotation mode. Instead of using the discrete base (see Chapter 6) $w_n = \arctan 2^{-n}$, 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:

$$\begin{cases} x_{n+1} = x_n - d_n 2^{-n} y_n + (1 - 2d_n^2) 2^{-2n-2} x_n \\ y_{n+1} = y_n + d_n 2^{-n} x_n + (1 - 2d_n^2) 2^{-2n-2} y_n \\ z_{n+1} = z_n - d_n w'_n = z_n - 2d_n \arctan 2^{-n-1}. \end{cases} \tag{7.14}$$

The new scaling factor is:

$$K_{\text{double}} = \prod_{i=1}^{\infty} \left(1 + 2^{-2i}\right) = 1.3559096738634793803 \cdots.$$

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 [300]. It consists of performing (7.14) with the following choice of d_n.

$$d_n = \begin{cases} -1 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] < 0 \\ 0 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] = 0 \\ +1 & \text{if } \left[z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right] > 0, \end{cases} \tag{7.15}$$

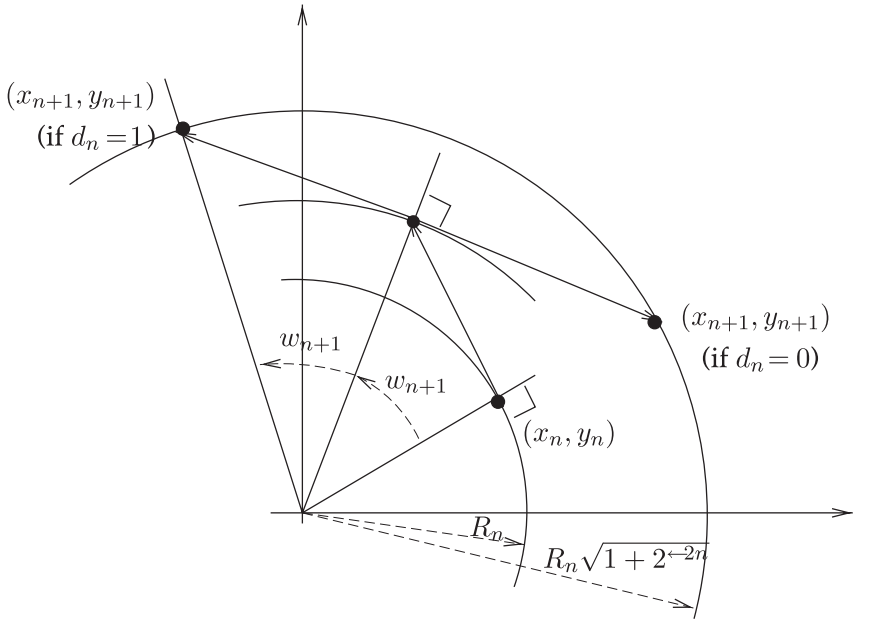


Figure 7.3: *One iteration of the double rotation method.*

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 (7.15) 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 $\left[z_n^{(n-2)} z_n^{(n-1)} z_n^{(n)} z_n^{(n+1)}\right]$ as we did, for instance, in Section 6.3.1.

Another redundant CORDIC method with constant scale factor, suggested by Takagi et al. [302], 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).

7.6 The Branching CORDIC Algorithm

This algorithm was introduced by Duprat and Muller [124]. Corrections and improvements have been suggested by Phatak [260, 261]. 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.

The algorithm uses a function "eval" such that,[4] at step n, we have:

$$\begin{cases} \text{eval}(z_n) \neq 0 \Rightarrow \text{sign}(z_n) = \text{eval}(z_n) \\ \text{eval}(z_n) = 0 \Rightarrow |z_n| \leq 2^{-n+1}. \end{cases} \tag{7.16}$$

The algorithm manipulates two values z_n, the one of module "$+$", named z_n^+, and the one of module "$-$", named z_n^-. One can show that:

- at least one of the terms $|z_n^+|$ and $|z_n^-|$ is less than or equal to

$$\sum_{k=n}^{\infty} \arctan 2^{-k};$$

- both terms are less than $3 \times 2^{-n+1}$.

The algorithm (extracted from [124]) is given in the following (in pseudo-Pascal). the variables zplus[n] and zminus[n] represent $|z_n^+|$ and $|z_n^-|$; and dplus[n] and dminus[n] denote the variable d_n of module "$+$" and module "$-$", respectively.

Algorithm 8 (Branching CORDIC)

```
Algorithm branching-CORDIC

procedure updatez(n);
  begin
    in parallel
         zplus[n+1] := zplus[n] - dplus[n] w[n];
        {in module "+"}
         zminus[n+1] := zminus[n] - dminus[n] w[n];
        {in module "-"}
```

[4] In the original paper [124], there was an error. Instead of the correct assumption

$$\text{eval}(z_n) = 0 \Rightarrow |z_n| \leq 2^{-n+1},$$

the authors assumed

$$\text{eval}(z_n) = 0 \Rightarrow |z_n| \leq 2^{-n-1}.$$

The correction was suggested by Phatak [260].

```
    in parallel
         splus := eval(zplus[n+1]);
         sminus := eval(zminus[n+1]);
  end;

begin
  i := 0;  {initializations}
  zplus[0] := zminus[0] := theta;
  splus := sminus := eval(theta);
1: while splus <> 0 and sminus <> 0 do
{while no branching}
    begin
       dplus[i] := splus;
       dminus[i] := sminus;
       updatez(i);
       i := i+1
    end;
2: {a branching is occurring}
    dplus[i] := 1;
    dminus[i] := -1;
    updatez(i);
    i := i+1;
3: while (splus = -1) and (sminus = +1) do
{while branching}
    begin
       dplus[i] := splus;
       dminus[i] := sminus;
       updatez(i);
       i := i+1
    end;
    {new branching, or end of branching}
   if splus = 0 then
{module "+" performed the good computation}
    begin
       zminus[i] := zplus[i];
       goto 2 {new branching}
    end
   else if splus = +1 then
{module "+" performed the good computation}
    begin
       zminus[i] := zplus[i];
       sminus := splus;
       goto 1 {branching terminated}
    end
   else if sminus = 0 then
{module "-" performed the good computation}
    begin
       zplus[i] := zminus[i];
       goto 2 {new branching}
    end
```

```
   else if sminus = -1 then
{module "-" performed the good computation}
    begin
       zplus[i] := zminus[i];
       splus := sminus;
       goto 1
{branching terminated}
    end;
end.
```

Now let us focus on the implementation of the function eval. Let z_n be z_n^+ or z_n^-. Relation (7.16) must be satisfied by eval(z_n). This is done as follows (in a way very similar to what we did in Section 6.3.1). Assume we use the radix-2 signed-digit number system for representing z_n (this would be very similar if the carry-save number system were used). Let z_n^* be $2^{n-1}z_n$ truncated after the radix point. We have:

$$\left| z_n^* - 2^{n-1} z_n \right| \leq 1.$$

Since $|z_n| < 3 \times 2^{-n+1}$, $|z_n^*|$ is less than or equal to 3. Now define $\widehat{z}_n^*$ as the 3-digit number obtained by suppressing the digits of z_n^* of a weight greater than or equal to 2^3. We have:

$$-7 \leq \widehat{z}_n^* \leq +7$$

$$|\widehat{z}_n^* - z_n^*| \text{ is a multiple of } 8;$$

therefore

- if $-7 \leq \widehat{z}_n^* \leq -5$, then $z_n^* = \widehat{z}_n^* + 8 \geq 1$; therefore $z_n \geq 0$. We must choose eval$(z_n) = 1$;
- $\widehat{z}_n^* = -4$ is an impossible case;
- if $-3 \leq \widehat{z}_n^* \leq -1$, then $z_n^* = \widehat{z}_n^*$; therefore $z_n \leq 0$. We must choose eval$(z_n) = -1$;
- if $\widehat{z}_n^* = 0$, then $-1 \leq 2^{n-1} z_n \leq +1$. We must choose eval$(z_n) = 0$;
- if $1 \leq \widehat{z}_n^* \leq 3$, then $z_n \geq 0$. We must choose eval$(z_n) = +1$;
- $\widehat{z}_n^* = 4$ is an impossible case;
- if $\widehat{z}_n^* \geq 5$, then $z_n \leq 0$. We must choose eval$(z_n) = -1$.

A comparison of some variants of the Branching CORDIC algorithm are provided in [288].

7.7 The Differential CORDIC Algorithm

This algorithm was introduced by Dawid and Meyr [95]. 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 7.2), 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

$$\begin{cases} |\hat{z}_{n+1}| = ||\hat{z}_n| - \arctan 2^{-n}| \\ d_{n+1} \ = d_n \times \text{sign}\,(\hat{z}_{n+1})\,. \end{cases} \tag{7.17}$$

Assume that $\hat{z}_n$ is represented in the binary signed-digit number system (radix 2, digits -1, 0, and 1). Relations (7.17) 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 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 (7.17) in a digit-pipelined fashion: as soon as we get digits of $|\hat{z}_n|$, 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 $|\hat{z}_{n+1}|$. $|\hat{z}_{n+1}|$ is generated from $|\hat{z}_n|$ with *on-line delay* 1: as soon as we get the ith digit of $|\hat{z}_n|$, we can compute the $i-1$st digit of $|\hat{z}_{n+1}|$. This is because adding a signed-digit binary number ($|\hat{z}_n|$) 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 [17]). 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 (7.17) — may be as large as the word length (if the only nonzero digit

[5]On-line arithmetic was introduced in 1977 by Ercegovac and Trivedi [137]. It requires the use of a redundant number system and introduces parallelism between sequential operations by overlapping them in a digit-serial fashion. See [130] for a survey.

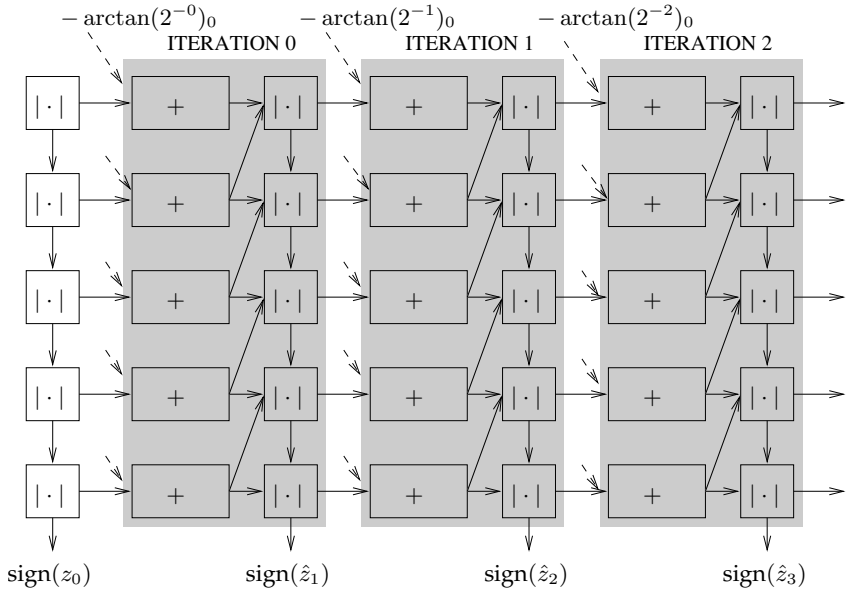


Figure 7.4: *Computation of the values sign($\hat{z}_i$) in the differential CORDIC algorithm (rotation mode) [95].*

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 7.4 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).

Algorithm 9 (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).*

$\hat{z}_0 = |z_0|\,, d_0 = \text{sign}\,(z_0)$
for $i = 0$ *to* n *do*

$$\begin{cases} |\hat{z}_{i+1}| = \big||\hat{z}_i| - \arctan 2^{-i}\big| \\ d_{i+1} \;\; = d_i \times \mathit{sign}(\hat{z}_{i+1}) \\ x_{i+1} \;\; = x_i - d_i y_i 2^{-i} \\ y_{i+1} \;\; = y_i + d_i x_i 2^{-i}. \end{cases}$$

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

$$\begin{cases} \hat{x}_{n+1} = \text{sign}(x_n) \times x_{n+1} \\ \hat{y}_{n+1} = \text{sign}(y_n) \times y_{n+1}. \end{cases}$$

The relations allowing us to find the sign of y_n are

$$\begin{cases} |\hat{y}_{n+1}| & = ||\hat{y}_n| - \hat{x}_n 2^{-n}| \\ \text{sign}(y_{n+1}) & = \text{sign}(\hat{y}_{n+1}) \times \text{sign}(y_n). \end{cases}$$

They can be implemented using an architecture very similar to that of Figure 7.4. Using that architecture, the following algorithm performs the CORDIC iterations in vectoring mode with a constant scale factor.

Algorithm 10 (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*

$$\begin{cases} |\hat{y}_{i+1}| = ||\hat{y}_i| - \hat{x}_i 2^{-i}| \\ d_{i+1} = d_i \times sign(\hat{y}_{i+1}) \\ \hat{x}_{i+1} = \hat{x}_i + |\hat{y}_i 2^{-i}| \\ z_{i+1} = z_i + d_i \arctan 2^{-i}. \end{cases}$$

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 $w_n = \ln(1 + 2^{-n})$ (see Chapter 6). Such a generalization is simple. Assume that we

[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.

want to get a restoring decomposition of a number x, $0 \leq x \leq \sum_{n=0}^{\infty} w_n$, that is, to get

$$x = \sum_{n=0}^{\infty} d_n w_n, d_n = 0, 1. \tag{7.18}$$

Defining $S = \sum_{n=0}^{\infty} w_n$, this gives $2x - S = \sum_{n=0}^{\infty}(2d_n - 1)w_n$, that is,

$$2x - S = \sum_{n=0}^{\infty} \delta_n w_n, \delta_n = \pm 1,$$

with $\delta_n = 2d_n - 1$.

Therefore, to get decomposition (7.18), it suffices to use the architecture described in Figure 7.4 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 $d_n = (\delta_n + 1)/2$.

7.8 Computation of $\cos^{-1}$ and $\sin^{-1}$ Using CORDIC

Now we present another application [227] of the double rotation method (see Section 7.5). 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:

$$\begin{cases} \theta_0 & = 0 \\ x_0 & = 1 \\ y_0 & = 0 \\ d_n & = \begin{cases} 1 & \text{if } \theta_n \leq \theta \\ -1 & \text{otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} & = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} & = \theta_n + d_n \arctan 2^{-n}, \end{cases} \tag{7.19}$$

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

$$d_n = \begin{cases} 1 & \text{if } \theta_n \leq \theta \\ -1 & \text{otherwise} \end{cases} \tag{7.20}$$

that appears in Eq. (7.19). However, (7.20) is equivalent to:

$$d_n = \begin{cases} \text{sign}(y_n) & \text{if } \cos(\theta_n) \geq \cos(\theta) \\ -\text{sign}(y_n) & \text{otherwise,} \end{cases} \tag{7.21}$$

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 $K_n = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}}$, (7.21) is equivalent to

$$d_n = \begin{cases} \text{sign}(y_n) & \text{if } x_n \geq K_n t \\ -\text{sign}(y_n) & \text{otherwise.} \end{cases} \tag{7.22}$$

If we assume that the values $t_n = K_n t$ are known, the algorithm

Algorithm 11 ($\cos^{-1}$: first attempt)

$$\begin{cases} \theta_0 & = 0 \\ x_0 & = 1 \\ y_0 & = 0 \\ d_n & = \begin{cases} \text{sign}(y_n) & \textit{if } x_n \geq t_n \\ -\text{sign}(y_n) & \textit{otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} & = \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} & = \theta_n + d_n \arctan 2^{-n} \end{cases} \tag{7.23}$$

gives $\theta_n \to_{n\to\infty} \cos^{-1} t$.

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 12 ($\cos^{-1}$ with double-CORDIC iterations)

$$\begin{cases} \theta_0 &= 0 \\ x_0 &= 1 \\ y_0 &= 0 \\ t_0 &= t \\ d_n &= \begin{cases} \text{sign}(y_n) & \textit{if } x_n \geq t_n \\ -\text{sign}(y_n) & \textit{otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} &= \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} &= \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} &= t_n + t_n 2^{-2n}. \end{cases} \tag{7.24}$$

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 13 ($\sin^{-1}$ with double-CORDIC iterations)

$$\begin{cases} \theta_0 &= 0 \\ x_0 &= 1 \\ y_0 &= 0 \\ t_0 &= t \\ d_n &= \begin{cases} \text{sign}(x_n) & \textit{if } y_n \leq t_n \\ -\text{sign}(x_n) & \textit{otherwise} \end{cases} \\ \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} &= \begin{pmatrix} 1 & -d_n 2^{-n} \\ d_n 2^{-n} & 1 \end{pmatrix}^2 \begin{pmatrix} x_n \\ y_n \end{pmatrix} \\ \theta_{n+1} &= \theta_n + 2d_n \arctan 2^{-n} \\ t_{n+1} &= t_n + t_n 2^{-2n}. \end{cases} \tag{7.25}$$

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 [198, 199] 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 (7.22) instead of $K_n t$ and easy to calculate.

7.9 Variations on CORDIC

Daggett [89] suggests the use of CORDIC for performing decimal-binary conversion. Schmid and Bogacki [277] 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 [194]. Radix-4 CORDIC algorithms are suggested by Antelo et al. [8, 251]. An algorithm for very high radix CORDIC rotation was suggested later by Antelo, Lang and Bruguera [9]. A software implementation of CORDIC is reported by Andrews and Mraz [7]. 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 [241]. Many other CORDIC chips are reported [78, 85, 110, 162, 297, 311, 326]. Some pipelined CORDIC architectures are suggested in [109, 189]. Adaptations of CORDIC to perform rotations in spaces of dimension higher than 2 are suggested by Delosme [107], Hsiao and Delosme [167], and Hsiao, Lau and Delosme [168]. 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 [171]. Wang and Swartzlander [323] suggest to pair-off some iterations to lower the hardware complexity of a CORDIC processor. Timmermann et al. propose an algorithm [309] that is based on Baker's prediction method (see Section 6.4). Adaptations of CORDIC to on-line arithmetic were suggested by Ercegovac and Lang [132, 133, 134], Lin and Sips [216, 217], Duprat and Muller [124], and Hemkumar and Cavallaro [164]. Kota and Cavallaro [192] show 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 [55], and by Hekstra and Deprettere [163]. An error analysis of CORDIC is given by Hu [170].

[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.

Chapter 8

Some Other Shift-and-Add Algorithms

8.1 High-Radix Algorithms

The shift-and-add algorithms presented in the previous chapters 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 [135], the price to pay is more complicated elementary iterations.

8.1.1 Ercegovac's radix-16 algorithms

The methods presented here are similar to methods suggested by Ercegovac [127]. They are generalizable to higher radices, different from 16. Some variants have been proposed by Xavier Merrheim [229]. Assume that we want to compute the exponential of t. To do this, we use a basic iteration very close to (6.15); that is:

$$\begin{aligned} L_{n+1} &= L_n - \ln\left(1 + d_n 16^{-n}\right) \\ E_{n+1} &= E_n \left(1 + d_n 16^{-n}\right), \end{aligned} \tag{8.1}$$

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 6.3.1, 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 [127], we choose $a = 10$.

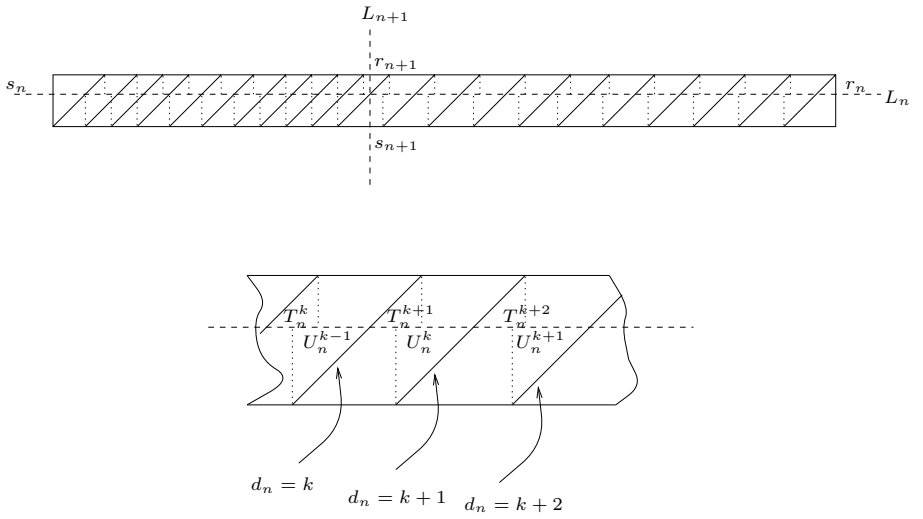


Figure 8.1: *Robertson diagram of the radix-16 algorithm for computing exponentials. T_k is the smallest value of L_n for which the value $d_n = k$ is allowable. U_k is the largest one.*

The Robertson diagram given in Figure 8.1 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 8.1 satisfy $r_{n+1} = r_n - \ln(1 + 10 \times 16^{-n})$ and $s_{n+1} = s_n - \ln(1 - 10 \times 16^{-n})$, which gives:

$$\begin{aligned} r_n &= \sum_{k=n}^{\infty} \ln\left(1 + 10 \times 16^{-k}\right) \\ s_n &= \sum_{k=n}^{\infty} \ln\left(1 - 10 \times 16^{-k}\right). \end{aligned} \tag{8.2}$$

For instance:

$$\begin{aligned} r_1 &= \sum_{n=1}^{\infty} \ln\left(1 + 10 \times 16^{-n}\right) \approx 0.526427859674 \\ s_1 &= \sum_{n=1}^{\infty} \ln\left(1 - 10 \times 16^{-n}\right) \approx -1.023282325006. \end{aligned} \tag{8.3}$$

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 8.1). We find:

$$\begin{aligned} T_n^k &= s_{n+1} + \ln(1 + k \times 16^{-n}) \\ U_n^k &= r_{n+1} + \ln(1 + k \times 16^{-n}). \end{aligned}$$

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 8.1: *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* $n \to \infty$.

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 8.1 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 8.1 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:

$$L_{n+1}^* = 16L_n^* - 16^{n+1} \ln\left(1 + d_n 16^{-n}\right). \tag{8.4}$$

The choice $d_n = k$ is allowable if and only if $16^n T_n^k \leq L_n^* \leq 16^n U_n^k$. One can show that for any $k \geq 2$,

$$16^n T_n^k < k < 16^n U_n^k.$$

This is illustrated by Table 8.2.

It is worth noticing that for $n \geq 3$, or $n = 2$ and $-8 \leq k \leq 8$, 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 $n \geq 3$, or $n = 2$ and $-8 \leq k \leq 8$, 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 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.

[1]More precisely, to the nearest integer in $\{-10, \ldots, 10\}$ if $n \geq 3$; in $\{-8, \ldots, 8\}$ if $n = 2$.

	$n=2$	$n=3$	$n=\infty$
$k=-10$	$[-10.87,-9.53]$	$[-10.68,-9.35]$	$\left[-10-\frac{2}{3},-10+\frac{2}{3}\right]$
$k=-9$	$[-9.83,-8.496]$	$[-9.68,-8.34]$	$\left[-9-\frac{2}{3},-9+\frac{2}{3}\right]$
$k=-8$	$[-8.80,-7.4618]$	$[-8.67,-7.34]$	$\left[-8-\frac{2}{3},-8+\frac{2}{3}\right]$
$k=-7$	$[-7.76,-6.43]$	$[-7.67,-6.34]$	$\left[-7-\frac{2}{3},-7+\frac{2}{3}\right]$
$k=-6$	$[-6.74,-5.41]$	$[-6.67,-5.34]$	$\left[-6-\frac{2}{3},-6+\frac{2}{3}\right]$
$k=-5$	$[-5.72,-4.38]$	$[-5.67,-4.34]$	$\left[-5-\frac{2}{3},-5+\frac{2}{3}\right]$
$k=-4$	$[-4.70,-3.37]$	$[-4.67,-3.34]$	$\left[-4-\frac{2}{3},-4+\frac{2}{3}\right]$
$k=-3$	$[-3.69,-2.35]$	$[-3.67,-2.33]$	$\left[-3-\frac{2}{3},-3+\frac{2}{3}\right]$
$k=-2$	$[-2.68,-1.34]$	$[-2.67,-1.33]$	$\left[-2-\frac{2}{3},-2+\frac{2}{3}\right]$
$k=-1$	$[-1.67,-0.36]$	$[-1.67,-0.33]$	$\left[-1-\frac{2}{3},-1+\frac{2}{3}\right]$
$k=0$	$[-0.67,0.67]$	$[-0.67,0.67]$	$\left[-\frac{2}{3},+\frac{2}{3}\right]$
$k=1$	$[0.33,1.66]$	$[0.33,1.67]$	$\left[1-\frac{2}{3},1+\frac{2}{3}\right]$
$k=2$	$[1.32,2.66]$	$[1.33,2.67]$	$\left[2-\frac{2}{3},2+\frac{2}{3}\right]$
$k=3$	$[2.32,3.65]$	$[2.33,3.67]$	$\left[3-\frac{2}{3},3+\frac{2}{3}\right]$
$k=4$	$[3.30,4.63]$	$[3.33,4.66]$	$\left[4-\frac{2}{3},4+\frac{2}{3}\right]$
$k=5$	$[4.28,5.62]$	$[4.33,5.66]$	$\left[5-\frac{2}{3},5+\frac{2}{3}\right]$
$k=6$	$[5.26,6.60]$	$[5.33,6.66]$	$\left[6-\frac{2}{3},6+\frac{2}{3}\right]$
$k=7$	$[6.24,7.57]$	$[6.33,7.66]$	$\left[7-\frac{2}{3},7+\frac{2}{3}\right]$
$k=8$	$[7.21,8.5434]$	$[7.33,8.66]$	$\left[8-\frac{2}{3},8+\frac{2}{3}\right]$
$k=9$	$[8.18,9.5113]$	$[8.32,9.66]$	$\left[9-\frac{2}{3},9+\frac{2}{3}\right]$
$k=10$	$[9.14,10.48]$	$[9.32,10.65]$	$\left[10-\frac{2}{3},10+\frac{2}{3}\right]$

Table 8.2: *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.*

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

$x \in$	ℓ
$[0, 1/32]$	0
$[1/32, 3/32]$	$\ln(1 + 2/32)$
$[3/32, 4/32]$	$\ln(1 + 4/32)$
$[4/32, 5/32]$	$\ln(1 + 5/32)$
$[5/32, 6/32]$	$\ln(1 + 6/32)$
$[6/32, 7/32]$	$\ln(1 + 7/32)$
$[7/32, 8/32]$	$\ln(1 + 9/32)$
$[8/32, 9/32]$	$\ln(1 + 10/32)$
$[9/32, 10/32]$	$\ln(1 + 11/32)$
$[10/32, 11/32]$	$\ln(1 + 13/32)$
$[11/32, 12/32]$	$\ln(1 + 14/32)$
$[12/32, 14/32]$	$\ln(1 + 16/32)$
$[14/32, 15/32]$	$\ln(1 + 19/32)$
$[15/32, 16/32]$	$\ln(1 + 20/32)$
$[16/32, 17/32]$	$\ln(1 + 22/32)$
$[17/32, 18/32]$	$\ln(1 + 24/32)$
$[18/32, 20/32]$	$\ln(1 + 26/32)$
$[20/32, 21/32]$	$\ln(1 + 29/32)$
$[21/32, 22/32]$	$\ln(1 + 31/32)$
$[22/32, 23/32]$	$\ln(2)$

Table 8.3: *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.*

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

$$M = e^{(2k+1)/32}.$$

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 8.3.

8.2 The BKM Algorithm

As shown in Chapter 7, 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 7), 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. [18], that allows us to perform rotations and to compute complex functions without scaling factors.

8.2.1 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 (7.1)). If we define a complex number E_n as $E_n = x_n + iy_n$, we obtain $E_{n+1} = E_n\,(1 + id_n 2^{-n})$, which is close to the basic step of Algorithm 6.1. This remark brings us to a generalization of that algorithm: we could perform multiplications by terms of the form $(1 + d_n 2^{-n})$, 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:

$$\begin{cases} E_{n+1} = E_n\,(1 + d_n 2^{-n}) \\ L_{n+1} = L_n - \ln\,(1 + d_n 2^{-n}) \end{cases} \tag{8.5}$$

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 6:

- 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.

8.2.2 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,

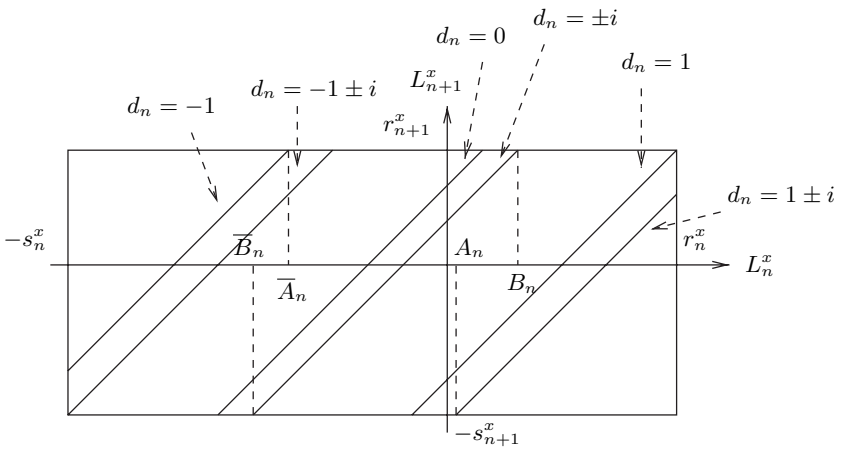


Figure 8.2: *The Robertson diagram for L^x_n [18].*

we find:

$$\begin{cases} L^x_{n+1} = L^x_n - \dfrac{1}{2}\ln\left[1 + d^x_n 2^{-n+1} + \left(d^{x\,2}_n + d^{y\,2}_n\right)2^{-2n}\right] \\ L^y_{n+1} = L^y_n - d^y_n \arctan\left(\dfrac{2^{-n}}{1 + d^x_n 2^{-n}}\right). \end{cases} \tag{8.6}$$

Now we give an algorithm that computes the sequence d_n for any L_1 belonging to a rectangular set $R_1 = [-s^x_1, r^x_1] + i\,[-r^y_1, r^y_1]$. The proof of the algorithm is based on the construction of a sequence $R_n = [-s^x_n, r^x_n] + i\,[-r^y_n, r^y_n]$ 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^x_n is chosen by examining a few digits of L^x_n and the "imaginary part" d^y_n is chosen by examining a few digits of L^y_n. These properties allow a simple and fast implementation of the choice of d_n.

Choice of d^x_n

The Robertson diagram presented in Figure 8.2 shows the different parameters involved in determining d^x_n. The diagram is constructed as follows.

- We assume that L^x_n belongs to the interval $[-s^x_n, r^x_n]$, which is the real part of R_n.
- L^x_{n+1} is equal to $L^x_n - (1/2)\ln\left[1 + d^x_n 2^{-n+1} + \left(d^{x\,2}_n + d^{y\,2}_n\right)2^{-2n}\right]$, so the value of L^x_{n+1} versus L^x_n is given by various straight lines parameterized by d^x_n and d^y_n.
- d^x_n must be such that *for any possible value of d^y_n*, $L^x_{n+1} \in \left[-s^x_{n+1}, r^x_{n+1}\right]$.

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 8.2 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 $L_n^x \in [-s_n^x, r_n^x]$, these choices will give a value of L_{n+1}^x between $-s_{n+1}^x$ and r_{n+1}^x:

$$\begin{cases} \text{if } L_n^x < -\bar{B}_n & \text{then } d_n^x = -1 \\ \text{if } -\bar{B}_n \le L_n^x < \bar{A}_n & \text{then } d_n^x = -1 \text{ or } 0 \\ \text{if } \bar{A}_n \le L_n^x < A_n & \text{then } d_n^x = 0 \\ \text{if } A_n \le L_n^x \le B_n & \text{then } d_n^x = 0 \text{ or } 1 \\ \text{if } B_n < L_n^x & \text{then } d_n^x = 1. \end{cases} \tag{8.7}$$

Choice of d_n^y

We use the relation

$$L_{n+1}^y = L_n^y - d_n^y \arctan\left(\frac{2^{-n}}{1 + d_n^x 2^{-n}}\right).$$

Figure 8.3 shows the Robertson diagram associated with the choice of d_n^y. We want our choice to be independent of the choice of d_n^x. From this, we deduce:

$$r_n^y = \sum_{k=n}^{\infty} \arctan\left(\frac{2^{-k}}{1 + 2^{-k}}\right).$$

The terms C_n and D_n appearing in the diagram are

$$C_n = -r_{n+1}^y + \arctan\left(\frac{2^{-n}}{1 - 2^{-n}}\right)$$

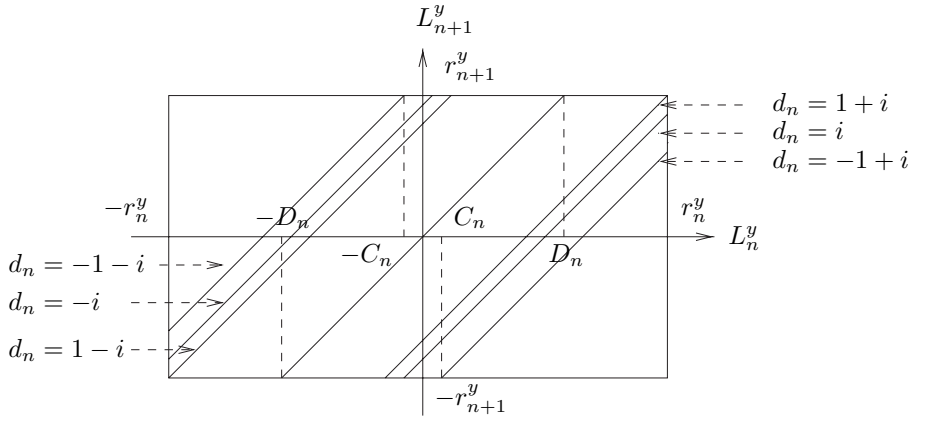


Figure 8.3: *The Robertson diagram for L^y_n [18].*

and

$$D_n = r^y_{n+1}.$$

One can prove that $C_n < D_n$. Thus, for any $L^y_n \in [-r^y_n, r^y_n]$, these choices will give a value of L^y_{n+1} between $-r^y_{n+1}$ and $+r^y_{n+1}$:

$$\begin{cases} \text{if } L^y_n < -D_n & \text{then } d^y_n = -1 \\ \text{if } -D_n \le L^y_n < -C_n & \text{then } d^y_n = -1 \text{ or } 0 \\ \text{if } -C_n \le L^y_n < C_n & \text{then } d^y_n = 0 \\ \text{if } C_n \le L^y_n \le D_n & \text{then } d^y_n = 0 \text{ or } 1 \\ \text{if } D_n < L^y_n & \text{then } d^y_n = 1. \end{cases} \tag{8.8}$$

The convergence domain R_1 of the algorithm is:

$$-0.8298023738\cdots = -s^x_1 \le L^x_1 \le r^x_1 = 0.8688766517\cdots$$
$$-0.749780302\cdots = -r^y_1 \le L^y_1 \le r^y_1 = 0.749780302\cdots.$$

The algorithm

Relations (8.7) and (8.8) 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: (8.7) and (8.8) 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:

$$\begin{cases} 2^n \bar{B}_n \le \bar{A} - 2^{-p_1} < \bar{A} \le 2^n \bar{A}_n \\ 2^n A_n \le A < A + 2^{-p_1} \le 2^n B_n \\ 2^n C_n \le C < C + 2^{-p_2} \le 2^n D_n. \end{cases} \tag{8.9}$$

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 (8.7) through (8.9):

- if $\tilde{L}_n^x \leq \bar{A} - 2^{-p_1}$, then $L_n^x \leq \bar{A}_n$; therefore $d_n^x = -1$ is a valid choice;
- if $\bar{A} \leq \tilde{L}_n^x \leq A$, then $\bar{B}_n \leq L_n^x \leq B_n$; therefore $d_n^x = 0$ is a valid choice[2];
- if $A + 2^{-p_1} \leq \tilde{L}_n^x$, then $A_n \leq L_n^x$; therefore $d_n^x = 1$ is a valid choice;
- if $\tilde{L}_n^y \leq -C - 2^{-p_2}$, then $L_n^y \leq -C_n$; therefore $d_n^y = -1$ is a valid choice;
- if $-C \leq \tilde{L}_n^y \leq C$, then $-D_n \leq L_n^y \leq D_n$; therefore $d_n^y = 0$ is a valid choice;
- if $C + 2^{-p_2} \leq \tilde{L}_n^y$, then $C_n \leq L_n^y$; therefore $d_n^y = 1$ is a valid choice.

Number of iterations

After n iterations of the E-mode, we obtain a relative error approximately equal to 2^{-n}.

8.2.3 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:

$$\lim_{n\to\infty} E_n = 1. \tag{8.10}$$

The following algorithm had been found through simulations before being proved. See [18] for more details.

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: $\begin{cases} E_{n+1} = E_n\,(1 + d_n 2^{-n}) \\ L_{n+1} = L_n - \ln\,(1 + d_n 2^{-n}) \end{cases}$
  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

$$\epsilon_n = 2^n(E_n - 1)$$

[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 $\tilde{L}_n^x \geq \bar{A}$.

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:

$$\begin{cases} \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } 6/16 \le \tilde{\epsilon}_1^y & \text{then } d_1 = 1 - i \\ \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } \tilde{\epsilon}_1^y \le -6/16 & \text{then } d_1 = 1 + i \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } 8/16 \le \tilde{\epsilon}_1^y & \text{then } d_1 = -i \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } \tilde{\epsilon}_1^y \le -9/16 & \text{then } d_1 = i \\ \text{if } \tilde{\epsilon}_1^x \le -7/16 \text{ and } |\tilde{\epsilon}_1^y| \le 5/16 & \text{then } d_1 = 1 \\ \text{if } -6/16 \le \tilde{\epsilon}_1^x \text{ and } |\tilde{\epsilon}_1^y| \le 1/2 & \text{then } d_1 = 0; \end{cases}$$

– 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 [18] for a proof of the algorithm.

8.2.4 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.

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 E-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 $\ln(x^2 + y^2)/2$.

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 $y/\sqrt{a}$.

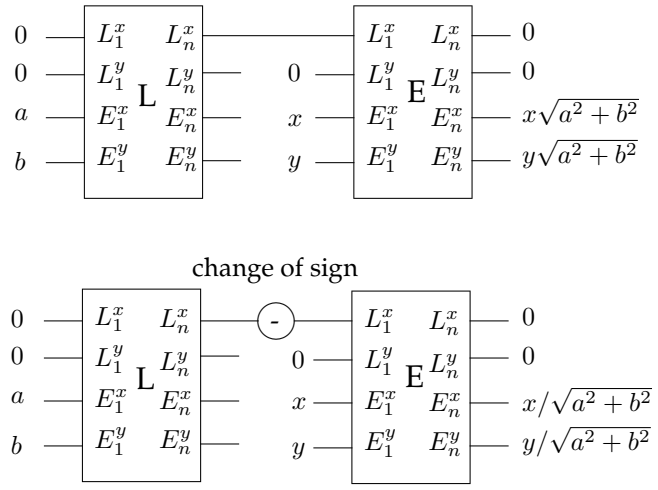


Figure 8.4: *Computation of lengths and normalization [18].*

- **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 e^F or e^{-F} (see Figure 8.4).

A generalization of BKM to radix 10 arithmetic was suggested by Imbert, Muller and Rico [175]. A High-Radix version of BKM was described by Didier and Rico [114].

Chapter 9

Range Reduction

9.1 Introduction

The algorithms presented in the previous chapters for evaluating the elementary functions only give a correct result if the argument is within a given bounded interval. To evaluate an elementary function $f(x)$ for any x, one must find some "transformation" 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.

In practice, there are two kinds of reduction:

- *Additive reduction.* x^* is equal to $x - kC$, where k is an integer and C 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 a constant (for instance, for the logarithm function, a convenient choice for C is the radix of the number system).

Example 9 (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* $g(x^*, k) =$

$$\begin{cases} \cos(x^*) & \text{if } k \bmod 4 = 0 \\ -\sin(x^*) & \text{if } k \bmod 4 = 1 \\ -\cos(x^*) & \text{if } k \bmod 4 = 2 \\ \sin(x^*) & \text{if } k \bmod 4 = 3; \end{cases} \tag{9.1}$$

- *obtain* $\cos(x) = g(x^*, k)$.

The previous reduction mechanism is an *additive* range reduction. Let us examine another example of additive reduction.

Example 10 (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 [307] uses

$$C = \frac{\ln(2)}{32}$$

so that the argument is reduced to the 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 double-precision arithmetic, exponentials of numbers less than $\ln(2^{-1074}) \approx -744.44$ are underflows, and exponentials of numbers larger than $\ln(2^{1024}) \approx +709.78$ are overflows.

Example 11 (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 mantissa, 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 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 it should be represented in a larger format, or by several (in most cases, two) machine numbers. 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 9.1 gives the computed value of $\sin(10^{22})$ on various computing systems (the figures in that table were picked up from [244, 245] and from various contributors belonging to the Computer Science Department of École Normale Supérieure de Lyon). 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 is actually approximated is not the sine function, but the function

$$\mathrm{fsin}(x) = \sin\left(\frac{\pi x}{fpi}\right)$$

where fpi is the floating-point number that is closest to π (for instance, in Table 9.1, the number $0.8740\cdots$ returned by the Silicon Graphics Indy computer is equal to $\mathrm{fsin}(10^{22})$). This makes range reduction easier, but there are 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 double-precision arithmetic, the machine number that is closest to $\mathrm{fsin}(1/4)$ is

$$\frac{1114208378708655}{4503599627370496}$$

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
DECstation 3100	NaN
Casio fx-8100, fx180p, fx 6910 G	Error
TI 89	Trig. arg. too large

Table 9.1: $\sin(x)$ *for* $x = 10^{22}$ *[244]. It is worth noticing that* x *is exactly representable in the IEEE-754 double-precision format (*10^{22} *is equal to* $4768371582031250 \times 2^{21}$*). With a system working in the IEEE-754 single-precision format, the correct answer would be the sine of the floating-point number that is closest to* 10^{22}*; that is,* $\sin(99999997781963083 61216) \approx -0.73408$*. As pointed out by the authors of [244, 245], the values listed in this table were contributed by various Internet volunteers, so they are not the official views of the listed computer system vendors, the author of [244, 245] or his employer, nor are they those of the author of this book.*

whereas the machine number that is closest to $\sin(1/4)$ is

$$\frac{4456833514834619}{18014398509481984}.$$

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 $\tan(x)$. In [68], Cody gives the example of the computation of $\sin(22)$ on many computing systems. Some of Cody's figures, along with new figures computed on current systems, are given in Table 9.2.

Computing System	$\sin(22)$
Exact value	$-8.85130929040388\text{e}{-3}$
TI 59	$-8.851309285516\text{e}{-3}$
TI 25	$-8.8487\text{e}{-3}$
TI 89	$-8.8513092904\text{e}{-3}$
HP 65	$-8.851306326\text{e}{-3}$
HP 34C	$-8.851309289\text{e}{-3}$
HP 48SX	$-8.8513092904\text{e}{-3}$
Casio FX-702P	$-8.851309219\text{e}{-3}$
PC: Borland Turbo Pascal 7.0	$-8.8513093008\text{e}{-3}$

Table 9.2: $\sin(x)$, *computed for* $x = 22$ *on several computing systems. Some of these figures are picked up from [68], the other ones have been computed on more recent systems.*

It is easy to understand why a bad range reduction algorithm gives inaccurate results. The naive method consists of performing the computations

$$\begin{aligned} k &= \left\lfloor \frac{x}{C} \right\rfloor \\ x^* &= x - kC \end{aligned}$$

using the machine precision. When kC is close to x, almost all the accuracy, if not all, is lost when performing the subtraction $x - kC$. For instance, on a radix-10 computer with 10 mantissa digits, 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^*.

A first solution consists of using multiple-precision arithmetic, but this may make the computation much slower. Moreover, it is not that easy to predict the precision with which the computation should be performed. After a first method due to Cody and Waite [64, 68], that works for rather small input arguments only but is very inexpensive, we present results due to Kahan [179] that allow us to find the worst cases for range reduction. Then we give two algorithms, one due to Payne and Hanek [258], another one due to Daumas et al. [93, 94] that make it possible to perform an accurate range reduction, without actually needing multiple-precision calculations.

9.2 Cody and Waite's Method for Range Reduction

The method suggested by Cody and Waite [64, 68] 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 $x - kC$, we evaluate[2]

$$(x - kC_1) - kC_2. \tag{9.2}$$

When doing this, if k is such that kC_1 is exactly representable, the subtraction $(x - kC_1)$ is performed without any error.[3] Therefore (9.2) 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 [64] are:

- in the IEEE-754 single-precision format:

$$C_1 = 201/64 = 3.140625$$
$$C_2 = 9.67653589793 \times 10^{-4};$$

- in the IEEE-754 double-precision format:

$$C_1 = 3217/1024 = 3.1416015625$$
$$C_2 = -8.089102067615373566 17 \times 10^{-6}.$$

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 double-precision CRLIBM library[4] [92] (see Section 12.4) uses four possible methods, 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-double arithmetic and a 64-bit integer format for k;
- Payne and Hanek's algorithm (see Section 9.4) for the largest arguments.

[1] In the radix of the floating-point system — 2 in general — not in radix 10.

[2] Gal and Bachelis [148] use the same kind of range reduction, but slightly differently. Some accuracy is lost when subtracting kC_2 from $x - kC_1$. To avoid this, they keep these terms separate. This is also done by Tang (see section 4.2.1).

[3] Unless subtraction is performed without any guard digit. To my knowledge, there is no current computer with such a poor arithmetic.

[4] `http://lipforge.ens-lyon.fr/projects/crlibm/`.

9.3 Finding Worst Cases for Range Reduction?

9.3.1 A few basic notions on continued fractions

To estimate the potential loss of accuracy that can occur when performing range reduction, we use the theory of continued fractions. We just recall here the elementary results that we need in the following. For more information, a good reference on continued fractions is Stark's book [291].

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):

$$\begin{cases} r_0 &= \alpha \\ a_i &= \lfloor r_i \rfloor \\ r_{i+1} &= \dfrac{1}{r_i - a_i}. \end{cases} \tag{9.3}$$

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 13 *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

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}}}$$

9.3.2 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-r number system, with n-digit mantissas, 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 a multiple of $\pi/8$ — 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:

$$x = x_0.x_1x_2x_3\cdots x_{n-1} \times r^{exponent}, \text{ with } x_0 \neq 0.$$

We can rewrite x with an integral mantissa:

$$x = M \times r^{exponent-n+1},$$

where $M = x_0x_1x_2x_3\cdots x_{n-1}$ satisfies $r^{n-1} \leq M \leq r^n - 1$. Performing the range reduction is equivalent to finding an integer p and a real number s, $|s| < 1$ such that

$$\frac{x}{C} = p + s. \tag{9.4}$$

The number x^* of the introduction of this chapter is equal to sC. We can rewrite (9.4) as

$$\frac{r^{exponent-n+1}}{C} = \frac{p}{M} + \frac{s}{M}.$$

If the result x^* of the range reduction is a very small number ϵ, this means that

$$\frac{r^{exponent-n+1}}{C} = \frac{p}{M} + \frac{\epsilon}{MC} \approx \frac{p}{M}. \tag{9.5}$$

In such a case, p/M must be a *very good rational approximation* to the irrational number $r^{exponent-n+1}/C$. To estimate the smallest possible value of ϵ, it suffices to examine the sequence of the "best" possible rational approximations to $r^{exponent-n+1}/C$, that is, the sequence of its convergents. 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 $r^{exponent-n+1}/C$. We then select the approximation P_j/Q_j whose denominator is the largest less than r^n. From Theorem 13, we know that for any rational number P/M, with $M \leq r^n - 1$, we have

$$\frac{M}{Q_j}\left|\frac{P}{M} - \frac{r^{exponent-n+1}}{C}\right| \geq \left|\frac{P_j}{Q_j} - \frac{r^{exponent-n+1}}{C}\right|.$$

Therefore

$$\left|\frac{P}{M} - \frac{r^{exponent-n+1}}{C}\right| \geq \frac{Q_j}{M}\left|\frac{P_j}{Q_j} - \frac{r^{exponent-n+1}}{C}\right|.$$

Therefore

$$MC\left|\frac{P}{M} - \frac{r^{exponent-n+1}}{C}\right| \geq CQ_j\left|\frac{P_j}{Q_j} - \frac{r^{exponent-n+1}}{C}\right|.$$

Hence the lowest possible value[5] of ϵ is attained for $M = Q_j$ and $P = P_j$, and is equal to

$$CQ_j \left| \frac{P_j}{Q_j} - \frac{r^{exponent-n+1}}{C} \right|.$$

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 [179] (a C-program that implements this method without needing extra-precision arithmetic was written by Kahan and McDonald). A similar method was found later by Smith [289]. The following Maple program implements this method.

```
worstcaseRR := proc(B,n,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-n)/C;
      for e from emin-n+1 to emax-n+1 do
          powerofBoverC := B*powerofBoverC;
          a := floor(powerofBoverC);
          Plast := a;
          r := 1/(powerofBoverC-a);
          a := floor(r);
          Qlast := 1;
          Q := a;
          P := Plast*a+1;
          while Q < B^n-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;
```

[5] If Q_j is less than r^{n-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 $r^{n-1} \leq r^\ell Q_j < r^n$; from

$$CQ_j \left| \frac{P_j}{Q_j} - \frac{r^{exponent-n+1}}{C} \right| = Cr^\ell Q_j \left| \frac{P_j}{r^\ell Q_j} - \frac{r^{exponent-\ell-n+1}}{C} \right|$$

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 r^{n-1} has no influence on the final result, that is, the search for the lowest possible value of ϵ.

r	n	C	e_{max}	Worst Case	$-\log_r(\epsilon)$
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	$614799 \cdots 1953734 \times 2^{+797}$	122.79

Table 9.3: *Worst cases for range reduction for various floating-point systems and reduction constants C.*

```
        epsilon :=
            evalf(C*abs(Plast-Qlast*powerofBoverC));
          if epsilon < epsilonmin then
            epsilonmin := epsilon; numbermin := Qlast;
            expmin := e
        fi
    od;
    print('mantissa',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 9.3. 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 9.3, the number $-\log_r(\epsilon)$ 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-r digits. Since x^* may be as small as ϵ, it must be computed with an absolute error less than $r^{-m-\log_r(\epsilon)}$. Roughly speaking, $-\log_r(\epsilon)$ is the number of "guard digits" that are necessary to perform the range reduction accurately.

By examining Table 9.3, we can see that $-\log_r(\epsilon)$ is always close to the number of digits of the number system (i.e., n 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 $1/r$ for each possible digit, then we can show that the most probable value for $-\log_r(\epsilon)$ is close to n plus the radix-r 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 10.5).

9.4 The Payne and Hanek Reduction Algorithm

We assume in the following that we want to perform the range reduction for the trigonometric functions, with $C = \pi/4$, and that the convergence domain of the algorithm used for evaluating the functions contains $I = [0, \pi/4]$. An adaptation to different values of C and I is quite straightforward.

From an input argument x, we want to find the reduced argument x^* and an integer k, that satisfy:

$$\begin{aligned} k &= \left\lfloor \frac{4}{\pi} x \right\rfloor \\ x^* &= \frac{\pi}{4}\left(\frac{4}{\pi}x - k\right). \end{aligned} \tag{9.6}$$

One can easily see that, once x^* is known, it suffices to know $k \bmod 8$ to deduce $\sin(x)$ or $\cos(x)$ from x^*. If x is large, or if x is very close to a multiple of $\pi/4$, the direct use of (9.6) to determine x^* may require the knowledge of $4/\pi$ with very large precision, and a cost-expensive multiple-precision computation if we wish the range reduction to be accurate.

Let us consider the following example. Assume a 16-bit mantissa, radix-2, floating-point number system, with rounding to the nearest. If $x = 1.011000000000000 \times 2^4 = 22_{10}$, then $(4/\pi)x = 11100.00000010111$ to the machine precision. Therefore $k = 28$ and $(4/\pi)x - k = 1.0111 \times 2^{-7}$. This gives $x^* = 1.001000010000010 \times 2^{-7}$, whose sine is $1.001000010000010 \times 2^{-7}$, whereas the sine of x is

$$-1.0010001000001 \times 2^{-7}.$$

We only have 8 significant bits!

Now let us present Payne and Hanek's reduction method [258, 289]. Assume an n-bit mantissa radix-2 floating point format (the number of bits n includes the possible hidden bit; for instance, with an IEEE double-precision format, $n = 53$). Let x be the floating-point argument to be reduced; let e be its unbiased exponent. One can write:

$$x = X \times 2^{e-n+1}$$

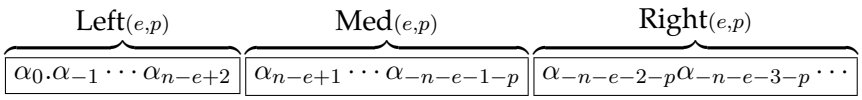


Figure 9.1: *The splitting of the digits of $4/\pi$ in Payne and Hanek's reduction method.*

where X is an n-bit integer satisfying $2^{n-1} \le X < 2^n$. We can assume $e \ge -1$ (since if $e < -1$, no reduction is necessary). Let

$$\alpha_0 \cdot \alpha_{-1}\alpha_{-2}\alpha_{-3}\alpha_{-4}\alpha_{-5}\cdots$$

be the infinite binary expansion of $\alpha = 4/\pi$. The first 1000 bits of α are:

1.01000101111100110000011011011100100111001000100000101010010100111111110000
10011101010111110100011111010100110100110111011100000011011011011000101001 0
10110011001001111000100001110010000010000011111111001010001011000111010101 1
11011110101110111100010101100001101101110010010001101110001110100100001001 0
01101110100101110000000000110010010010010111011101010000010011101000110010 0
10000111001111111000011101111010110001110010110001001010011010011100111110 1
11010001000001000110101111101010010111010111011010001001000010011101001100 1
11000111000000100110101101000101111101111100100000100111001100100011101011
00011100110000011010100110011100111110100100111001000010001011111100010111 0
11110111111001001010000011101100011111111111000100101111111111111101111000000
10110011000000011111110111100101111000100011000101101011010000010100110110 1
00011111011011010011011001111110110011110010011111001011000010011011011101 0
01111010001100011111101100110100111100101111111101010001011010111010100100 1
11101110101100011111101011

Now let us introduce an integer parameter p (that is used to specify the required accuracy of the range reduction). One can rewrite $\alpha = 4/\pi$ as[6]

$$\text{Left}(e,p) \times 2^{n-e+2} + (\text{Med}(e,p) + \text{Right}(e,p)) \times 2^{-n-e-1-p},$$

where

$$\begin{cases} \text{Left}(e,p) &= \alpha_0\alpha_{-1}\cdots\alpha_{n-e+2} \\ \text{Med}(e,p) &= \alpha_{n-e+1}\alpha_{n-e}\cdots\alpha_{-n-e-1-p} \\ \text{Right}(e,p) &= 0.\alpha_{-n-e-2-p}\alpha_{-n-e-3-p}\alpha_{-n-e-4-p}\alpha_{-n-e-5-p}\cdots. \end{cases}$$

Figure 9.1 shows this splitting of the binary expansion of α.

For instance, if $n = 16, p = 10$, and $e = 31$, the numbers Left(e,p), Med(e,p), and Right(e,p) are:

$$\begin{cases} \text{Left}(e,p) &= 10100010111110 \\ \text{Med}(e,p) &= 01100000110110111011011111110111000100001 1010 \\ \text{Right}(e,p) &= 0.00111011010100000100110101000101001\cdots. \end{cases}$$

[6]When $n - e + 2 \ge 0$, Left(e,p) does not exist, and the splitting of α starts with Med(e,p).

The basic idea of the Payne–Hanek reduction method is to notice that, if p is large enough, $\mathrm{Med}(e,p)$ contains the only digits of $\alpha = 4/\pi$ that matter for the range reduction. Since

$$\frac{4}{\pi}x = \mathrm{Left}(e,p) \times X \times 8 + \mathrm{Med}(e,p) \times X \times 2^{-2n-p} + \mathrm{Right}(e,p) \times X \times 2^{-2n-p},$$

the number $\mathrm{Left}(e,p) \times X \times 8$ is a multiple of 8, so that once multiplied by $\pi/4$ (see Eq. (9.6)), it will have no influence on the values of the trigonometric functions. $\mathrm{Right}(e,p) \times X \times 2^{-2n-p}$ is less than 2^{-n-p}; therefore it can be made as small as desired. For instance, if we want a reduced argument with at least m significant bits, and if the number $-\log_2(\epsilon)$ found using the algorithm presented in Section 9.3.2 is less than some integer j, then $p = j + m - n$ is convenient.

Payne and Hanek's algorithm is accurate and efficient even for large arguments. It has been widely used, for instance in some of Sun Microsystems libraries [244]. Now let us examine an example of a range reduction using Payne and Hanek's algorithm.

Example 12 (Payne and Hanek's algorithm) *Assume that we want to evaluate the sine of $x = 1.1_2 \times 2^{200}$, and that we use an IEEE-754 double-precision arithmetic. Also assume that we choose $p = 20$. We get the following values:*

$$\begin{aligned} n &= 53 \\ e &= 200 \\ X &= x \times 2^{n-1-e} = 6755399441055744 \\ \mathit{Med}(e,p) &= 22967545302678233671267846710948985065 9. \end{aligned}$$

Define h as $\mathit{Med}(e,p) \times X \times 2^{-2n-p}$. We have

$$h = 18238375864616442.953781343111871\cdots,$$

therefore $\lfloor h \rfloor \mod 8 = 2$, therefore $\sin(x)$ is approximated by $\cos(\mathit{frac}(h) \times \pi/4)$. The approximation is good: if we compute $\cos(\mathit{frac}(h) \times \pi/4)$, we get

$$0.732303330876108523957991\cdots,$$

whereas $\sin(x)$ is equal to

$$0.732303330876108523957972\cdots.$$

Now, assume that we want to evaluate the cosine of

$$x = 6381956970095103 \times 2^{797}$$

on the same arithmetic, with $p = 20$. This is the worst case for double-precision (see Table 9.3). We get the following parameters:

$$\begin{aligned} n &= 53 \\ e &= 849 \\ X &= x \times 2^{n-1-e} = 6381956970095103 \\ \mathit{Med}(e,p) &= 24142441824950440361333283508490629 6214. \end{aligned}$$

The number $h = \mathit{Med}(e,p) \times X \times 2^{-2n-p}$ *is equal to*

$$18111549684342730.000000000000000000596757467738309$$
$$107351120472437\cdots,$$

therefore $\lfloor h \rfloor \mod 8 = 2$, *therefore* $\cos(x)$ *is approximated by* $-\sin(\mathit{frac}(h) \times \pi/4)$. *The approximation is inaccurate: if we compute* $-\sin(\mathit{frac}(h) \times \pi/4)$, *we get*

$$-4.68692219155\cdots \times 10^{-19},$$

whereas $\cos(x)$ *is equal to*

$$-4.68716592425462761112\cdots \times 10^{-19}.$$

A larger value of p *must be chosen. If we perform the same computation with* $p = 60$, *we get a much better approximation of* $\cos(x)$*:*

$$-\sin(\mathit{frac}(h) \times \pi/4) = -4.6871659242546274384634\cdots \times 10^{-19}.$$

9.5 The Modular Range Reduction Algorithm

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:

$$x^* = x - kC. \tag{9.7}$$

If $\epsilon > 0$, then x^* and k are not uniquely defined by Eq. 9.7. 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 the usual algorithms for evaluating the elementary functions, one can assume that the length of the convergence domain I is greater than C, that is, that we can perform a *redundant* range reduction. For instance, with the CORDIC algorithm, when performing rotations (see Chapter 7), 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 present here an algorithm proposed by Daumas et al. [93, 94]. This algorithm is called the *modular range reduction algorithm* (MRR).

9.5.1 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:

$$x_{N-1}x_{N-2}x_{N-3}\cdots x_0.x_{-1}x_{-2}\cdots x_{-p}, \text{ where } x_i \in \{0,1\}$$

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 ν the integer such that $2^\nu < C \leq 2^{\nu+1}$.

Let us define, for $i \geq \nu$, 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.

First reduction We compute the number[7]:

$$\begin{aligned} r = (x_{N-1}m_{N-1}) + (x_{N-2}m_{N-2}) + \cdots + (x_\nu m_\nu) \\ + x_{\nu-1}x_{\nu-2}x_{\nu-3}\cdots x_0.x_{-1}x_{-2}\cdots x_{-p}. \end{aligned} \tag{9.8}$$

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.

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

$$m = \lfloor \log_2((N-\nu+2)C/2) \rfloor + \lceil -\log_2(\epsilon) \rceil$$

[7] This formula looks correct only for positive values of ν. It would be more correct, although maybe less clear, to write: $r = \sum_{i=\nu}^{N-1} x_i m_i + \sum_{i=-p}^{\nu-1} x_i 2^i$.

is very small in all practical cases (see the following example). If we define k as[8] $\lfloor \hat{r}/C \rceil$ (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 $|k - \hat{r}/C| \leq 1/2$; therefore $|\hat{r} - kC| \leq C/2$. From the definition of $\hat{r}$, we have

$$|r - \hat{r}| \leq 2^{\lfloor \log_2(\epsilon) \rfloor} \leq \epsilon;$$

therefore:

$$|r - kC| \leq \frac{C}{2} + \epsilon.$$

2. *In the positive case.* We have $k \leq \hat{r}/C < k + 1$; therefore $0 \leq \hat{r} - kC < C$; therefore $-\epsilon \leq r - kC < C + \epsilon$.

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 13 *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 7). 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.*

[8] We denote $\lfloor x \rceil$ the integer that is nearest to x.

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*

$$\sin(355) = -0.000030144353359488449\cdots.$$

9.5.2 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 the 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 9.3.2 is less than some integer j, then $q = j + t - 1 + \lceil \log_2(n+1) \rceil$ is convenient.

9.5.3 Architectures for modular reduction

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 [37], [88], [156], [240], [303], [319]), 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 [32], or merely a modified Booth recoding [174], 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 [209].

9.6 Alternate Methods

Defour, Kornerup, Muller and Revol [46, 104] 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 argument is split into eight 8-bit parts, and each part is used to address tables of double-precision numbers.

Li, Boldo and Daumas [213] give conditions that help to design efficient range reduction algorithms when a fused multiply-add instruction is available.

Chapter 10

Final Rounding

10.1 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. [287], monotonicity failures can cause 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 (see [103] for suggestions on what could be put in a 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 arithmetic. The machine number which is closest to $\pi/2$ is

$$\ell = \frac{13176795}{8388608} = 1.57079637050628662109375 > \frac{\pi}{2}.$$

If the arctangent function is implemented in round-to-nearest mode, then the computed value of the arctangent of any number greater than or equal to

[1] For any rounding mode, if the "exact" function is monotonic, and if exact rounding is provided, then the "computed function" is monotonic too.

[2] Correct rounding preserves symmetry if we round to the nearest or towards zero, that is, if the rounding function itself is symmetrical.

62919776 will be ℓ, and therefore will be larger than $\pi/2$. A consequence of that is that for any single-precision number $x \geq 62919776$,

$$\tan\left(\arctan(x)\right) = -22877332 = \tan(\ell).$$

In this chapter, we want to implement a function f in a radix-2 floating-point number system, with n mantissa bits, 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 made. After that, this intermediate result is rounded to the n-bit-mantissa target format, according to the active rounding mode. 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.

10.2 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

$$f\left(x + \mathrm{ulp}(x)\right) > f(x).$$

The computed value $F\left(x + \mathrm{ulp}(x)\right)$ is larger than or equal to $f\left(x + \mathrm{ulp}(x)\right) - \epsilon$, and $F(x)$ is less than or equal to $f(x) + \epsilon$. Therefore, to make sure that

$$F\left(x + \mathrm{ulp}(x)\right) \geq F(x)$$

it is sufficient that $f\left(x + \mathrm{ulp}(x)\right) - f(x) \geq 2\epsilon$. There exists a number $\xi \in [x, x + \mathrm{ulp}(x)]$ such that

$$f\left(x + \mathrm{ulp}(x)\right) - f(x) = \mathrm{ulp}(x) \times f'(\xi).$$

Therefore if

$$\epsilon \leq \frac{1}{2}\mathrm{ulp}(x) \times \min_{t \in [x, x+\mathrm{ulp}(x)]} \left|f'(t)\right|, \tag{10.1}$$

then the obtained implementation of f will be monotonic. Let us examine an example.

Example 14 (Monotonic evaluation of sines)
Assume that we want to evaluate the sine function on $[-\pi/4, +\pi/4]$. *If* $f(x) = \sin(x)$, *then*

$$\min_{[-\pi/4, +\pi/4]} \left|f'\right| = \cos\frac{\pi}{4} = 0.707\cdots.$$

[3] Computed with a precision somewhat larger than the "target precision."

Therefore if

$$\epsilon \leq \frac{1}{2} \times ulp(x) \times 0.707\cdots,$$

then condition (10.1) is satisfied. For $x \in [-\pi/4, +\pi/4]$*, we obviously have*

$$\frac{1}{2} \leq \frac{ulp(\sin(x))}{ulp(x)} \leq 1.$$

Therefore if

$$\frac{\epsilon}{ulp(\sin(x))} \leq \frac{1}{2} \times 0.707,$$

then condition (10.1) 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* $n+2$ *bits of precision will necessarily be monotonic when rounded to* n *bits of precision [141].*

Ferguson and Brightman [141] gave techniques that can be used to prove that an approximation is monotonic. Their main result is the following theorem.

Theorem 14 (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* $[a, b]$*.*

10.3 Correct Rounding: Presentation of the Problem

We assume that from any real number x and any integer m (with $m > n$), we are able to compute an approximation to $f(x)$ with an error in its mantissa y less than or equal to 2^{-m}. The computation can be carried out using a larger fixed-point or floating-point format, with one of the algorithms presented in this book.

Therefore the problem is to get an n-bit floating-point correctly rounded result from the mantissa 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,

$$\underbrace{1.\underbrace{xxxxx\cdots xxx}_{n \text{ bits}}}^{m \text{ bits}} 1000000\cdots 000000\, xxx\cdots$$

or

$$\underbrace{1.\underbrace{xxxxx\cdots xxx}_{n \text{ bits}}0111111\cdots111111}^{m \text{ bits}}xxx\cdots;$$

- in rounding towards $+\infty$, or towards $-\infty$ modes,

$$\underbrace{1.\underbrace{xxxxx\cdots xxx}_{n \text{ bits}}0000000\cdots000000}^{m \text{ bits}}xxx\cdots$$

or

$$\underbrace{1.\underbrace{xxxxx\cdots xxx}_{n \text{ bits}}1111111\cdots111111}^{m \text{ bits}}xxx\cdots.$$

This problem is known as the *Table Maker's Dilemma* (TMD) [152].

Let us denote $\diamond$ the rounding function. We will call a *breakpoint* a value z where the rounding 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 "directed" rounding modes (i.e., towards $+\infty$, $-\infty$ or 0), the breakpoints are the floating-point numbers. For rounding to the nearest mode, 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 mantissa,

$$\sin(11.1010) = 0.0\boxed{111011}01111110\cdots,$$

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 double-precision range [208] is attained for

$$x = 1.01100010101010001000011000010011011000101001010110110 \times 2^{678}$$

whose logarithm is

$$\log x = \overbrace{111010110.01000111100111101\cdots110001}^{53 \text{ bits}}\underbrace{000000000000000\cdots000000000000000}_{65 \text{ zeros}}1110\cdots$$

This is a "difficult case" in a directed rounding mode, 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 [208] is

$$1.11100100010110010110010100100110101111111001010010011101 \times 2^{-10}$$

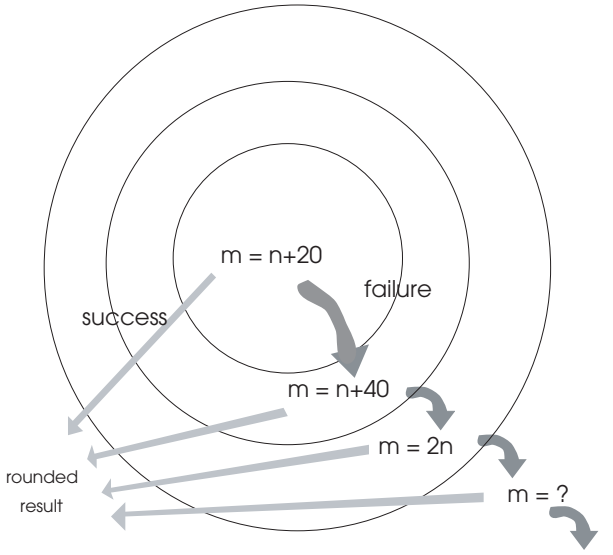


Figure 10.1: *Ziv's multilevel strategy.*

whose radix-2 exponential is

$$\overbrace{1.00000000010100111111111000010111\cdots0011}^{53\text{ bits}}$$
$$0\underbrace{11111111111111111\cdots11111111111111111}_{59\text{ ones}}0100\cdots$$

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" [328], illustrated in Figure 10.1, consists of starting to approximate the function with error $\pm 2^{-m_0}$, where m_0 is small but larger than n (say, $m_0 \approx n+10$ or $n+20$). 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, 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 12.3). 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 to know the way in which the process can be slow (i.e., to estimate the maximum possible value of m, if there is any).

In 1882, Lindemann showed that the exponential of an algebraic number (possibly complex) different from zero is not algebraic [19]. The machine

[4]The probability of a failure is about one over one million with $m_0 = n + 20$.

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.[5] Therefore, 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 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.

10.4 Some Experiments

Schulte and Swartzlander [281, 282] proposed algorithms for producing correctly rounded results for the functions $1/x$, square root, 2^x, and $\log_2 x$ in single-precision. To find the correct value of m, they performed an exhaustive search for $n = 16$ and $n = 24$. For $n = 16$, they found $m = 35$ for $\log_2$ and $m = 29$ for 2^x, and for $n = 24$, they found $m = 51$ for $\log_2$ and $m = 48$ for 2^x. One would like to extrapolate those figures and find $m \approx 2n$. To check that assumption, an exhaustive search was started, for some functions and domains, assuming double-precision arithmetic, in the Arénaire project of LIP laboratory (Lyon, France), using algorithms designed by Lefèvre [205, 206]. Before giving the obtained results, let us explain why in practice we always find that the required value of m is around $2n$ (or slightly above $2n$).

10.5 A "Probabilistic" Approach to the Problem

The approach chosen 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 $m \approx 2n$. 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 [119] and by Gal and Bachelis [148]. Stehlé and Zimmermann [293] also use this kind of probabilistic argument to implement a variant of Gal's Accurate Tables Method (see Chapter 4).

Let f be an elementary function. We assume in the following that when x is an n-bit mantissa floating-point number, the bits of $f(x)$ after the nth position

[5] This is not always true for some other functions, such as x^y.

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 [138], 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 (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 (see Tables 10.3 and 10.4). We also assume that the bit patterns (after position n) 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 4 to estimate the cost of building "accurate tables." When we compute the mantissa y of $f(x)$, the result has the form:

$$y = y_0.y_1y_2\cdots y_{n-1}\overbrace{01111111\cdots11}^{kbits}xxxxx\cdots$$

or

$$y = y_0.y_1y_2\cdots y_{n-1}\overbrace{10000000\cdots00}^{kbits}xxxxx\cdots$$

with $k \geq 1$. What we want to estimate is the maximum possible value of k. That value, added with n, 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 the input numbers are n-mantissa bit normalized floating-point numbers, and that there are n_e different possible exponents. Therefore there are $N = 2 \times n_e \times 2^{n-1}$ 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:

$$\mathcal{P}_{k_0} = 1 - \left[1 - 2^{1-k_0}\right]^N. \tag{10.2}$$

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

$$\ln\frac{1}{2} \leq N \times \ln\left[1 - 2^{1-k_0}\right]. \tag{10.3}$$

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 Eq. (10.3) 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:

$$k_0 \geq 1 - \frac{\ln(\ln 2)}{\ln 2} + \log_2 N \simeq 1.529 + \log_2 N;$$

	k										
n	1	2	3	4	5	6	7	8	9	10	11
1	1	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0
3	1	2	0	1	0	0	0	0	0	0	0
4	3	3	1	1	0	0	0	0	0	0	0
5	9	4	0	2	1	0	0	0	0	0	0
6	16	7	6	2	1	0	0	0	0	0	0
7	33	14	8	3	2	2	2	0	0	0	0
8	67	27	16	11	5	1	0	0	0	0	1
9	132	57	34	20	8	1	1	1	0	2	0
10	255	128	71	27	15	7	4	2	2	0	1
11	506	265	113	62	35	18	15	4	3	1	2
12	1012	528	232	153	54	29	21	7	8	4	0
13	2049	1046	449	267	131	79	39	16	14	3	2
14	4080	2072	966	517	287	146	65	24	18	7	2
15	8160	4087	2152	994	502	247	133	53	28	13	8
16	16397	8151	4191	2043	1010	463	255	131	62	35	16
17	32764	16350	8170	4168	2080	1036	473	228	127	72	31

Table 10.1: *Number of occurrences of various k for* $\sin(x)$.

that is,

$$k_0 \geq n + \log_2(n_e) + 1.529. \tag{10.4}$$

Therefore if we only consider a small number of values of n_e (which is the case in practice for many elementary functions[6]), the maximum value of k encountered will be (on average) slightly greater than n. Since the value of m that must be chosen is equal to n plus the maximum value of k, we deduce that in practice, m must be slightly greater than $2n$. The probabilistic assumptions that we used cannot be proved, but we can try to check them. Table 10.1 shows the number of occurrences of the various possible values of k for $\sin x$ (with $1 \leq x < 2$) and n between 1 and 17. In each row, one can easily recognize numbers that are close to the powers of 2; this tends to show that our probabilistic arguments are realistic.

Until we get sure bounds[7] on m, our probabilistic assumptions can help to design algorithms that are very likely to always return correctly rounded

[6]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.

[7]For IEEE-754 double-precision, we have obtained the bounds for many functions and domains (see Section 10.7.2). For double-extended precision, the bounds should be obtained within a few years. Higher precisions seem out of reach.

results. Let us examine two examples.

- If we want to evaluate exponentials of IEEE-754 double-precision floating-point numbers, 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^{-54}, then one can easily deduce the value that must be returned (1 in round-to-nearest mode, $1 + \mathrm{ulp}(1)$ if $x > 0$ in round towards $+\infty$ mode... see Tables 10.3 and 10.4). Therefore we must consider 64 possible exponents. Thus the number N of floating-point numbers to be considered in (10.2) is $2 \times 64 \times 2^{52}$. If we apply (10.2), we find that

  - 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 10.5). For double-precision 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 double-precision 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^{-27} one can easily deduce the values that must be returned (see Tables 10.3 and 10.4). Therefore we must consider 1051 possible exponents. Thus N equals $2 \times 1051 \times 2^{52}$. If we apply (10.2), 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 double-precision range. We only know worst cases for input values less than $12867/8192 \approx 1.5706787$ (see Table 10.11) for the cosine function, and less than $2 + 4675/8192 \approx 2.5707$ (see Table 10.9) for the sine function.

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."

n	Range for exp	m
24	$\ln 2$	494416
	10	3074888
53	$\ln 2$	1038560
	10	5234891
112	$\ln 2$	2527507
	10	10409113

Table 10.2: *Upper bounds on m for various values of n.*

10.6 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 single-precision. Such an exhaustive search is almost finished for double-precision [210, 208] (see Section 10.7), and has recently been launched for double-extended precision [292]. However, for larger precisions (e.g., quad-precision), an exhaustive search is far beyond the abilities of the most powerful current computers. Only theoretical results could allow us to manage such cases. To the knowledge of the author, the best current result is the following [242].

Let p/q be a rational number, with $q > 0$ and $\gcd(p, q) = 1$. Define $H(p/q) = \max\{|p|, q\}$.

Theorem 15 (Y. Nesterenko and M. Waldschmidt (1995)) *Let α, β be rational numbers. Let A, B, and E be positive real numbers with $E \geq e$ satisfying*

$$A \geq \max\left(H(\alpha), e\right), \quad B \geq H(\beta).$$

Then

$$\begin{aligned}&|e^{\beta} - \alpha| \geq \\ &\exp\Big(-211 \times (\ln B + \ln\ln A + 2\ln(E|\beta|_+) + 10) \\ &\times (\ln A + 2E|\beta| + 6\ln E) \times (3.3\ln(2) + \ln E) \times (\ln E)^{-2}\Big),\end{aligned}$$

where $|\beta|_+ = \max(1, |\beta|)$.

We can apply this theorem with α and β being machine numbers. By doing this, we get a lower bound on the distance between a machine number and the exponential of a machine number (or a bound on the distance between a machine number and the logarithm of a machine number). This is exactly what we need. Table 10.2 gives the values of m obtained using this theorem, for various values of n.

As a consequence, in the very worst cases, we may have to compute exponentials or logarithms with a number of bits whose order of magnitude is a

few millions (around 10 millions for exponentials of quad-precision numbers less than 10). We must keep in mind that, following our probabilistic study, the existence of such cases is *extremely unlikely*. Using algorithms based on the arithmetic-geometric mean (AGM) (See Chapter 5), computing exponentials with a few millions of bits would require a few minutes on current computers.

10.7 Obtained Worst Cases for Double-Precision

The previous section may seem disappointing. For instance, in double-precision arithmetic, although we are almost certain (from our probabilistic estimates) that it suffices to evaluate the functions with an intermediate accuracy that is around 120 bits, it seems that to be able to certify that we always return a correctly rounded result we must evaluate the functions[8] with millions of bits. The only way to solve this problem is to actually compute the worst cases (that is, the hardest to round cases), at least for the most common functions and domains.

10.7.1 Special input values

For most 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 floating-point format with n-bit mantissas, assuming rounding to nearest. If $\epsilon < 2^{-n}$, then (since ϵ is an n-bit number), $\epsilon \leq 2^{-n} - 2^{-2n}$. Hence,

$$e^{\epsilon} \leq 1 + (2^{-n} - 2^{-2n}) + \frac{1}{2}(2^{-n} - 2^{-2n})^2 + \cdots < 1 + 2^{-n},$$

therefore

$$\exp(\epsilon) < 1 + \frac{1}{2}\ \mathrm{ulp}\ (\exp(\epsilon)).$$

Thus, 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 10.3 and 10.4.

10.7.2 Lefèvre's experiment

In his PhD dissertation [206], Lefèvre gives algorithms for finding the worst cases of the table maker's dilemma. These algorithms use linear approximations and massive parallelism. A recent presentation of these algorithms, with some improvements, can be found in [207]. We have run Lefèvre's algorithms to find worst cases in double-precision for the most common functions and domains. The results obtained so far, first published in [208] are given in Tables 10.5 to 10.14.

[8] In the worst case: we can use Ziv's strategy.

This function	can be replaced by	when
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-53}$
$\exp(\epsilon), \epsilon \leq 0$	1	$\lvert\epsilon\rvert \leq 2^{-54}$
$\ln(1+\epsilon)$	ϵ	$\lvert\epsilon\rvert < \sqrt{2} \times 2^{-53}$
$2^{\epsilon}, \epsilon \geq 0$	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	$\lvert\epsilon\rvert < 1.7368 \times 2^{-56}$
$\sin(\epsilon), \arcsin(\epsilon), \sinh(\epsilon), \sinh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq 1.4422\cdots \times 2^{-26}$
$\cos(\epsilon)$	1	$\lvert\epsilon\rvert < \sqrt{2} \times 2^{-27}$
$\cosh(\epsilon)$	1	$\lvert\epsilon\rvert < 2^{-26}$
$\tan(\epsilon), \tanh(\epsilon), \arctan(\epsilon), \tanh^{-1}(\epsilon)$	ϵ	$\lvert\epsilon\rvert \leq 1.817\cdots \times 2^{-27}$

Table 10.3: *Some results for small values in double-precision, assuming **rounding to the nearest**. These results make finding worst cases useless for negative exponents of large absolute value.*

If a and b belong to the same "binade" (they have the same sign and satisfy $2^p \leq |a|, |b| < 2^{p+1}$, where p is an integer), let us call their *mantissa distance* the distance

$$\frac{|a-b|}{2^p}.$$

For instance, the mantissa distance between 14 and 15 is $(15-14)/8 = 1/8$. Tables 10.5 to 10.14 allow to deduce properties such as the following ones [208].

Theorem 16 (Computation of exponentials) *Let y be the exponential of a double-precision number x. Let y^* be an approximation to y such that the mantissa distance[9] between y and y^* is bounded by ϵ.*

- *for $|x| \geq 2^{-30}$, if $\epsilon \leq 2^{-53-59-1} = 2^{-113}$, then for any of the four rounding modes, rounding y^* is equivalent to rounding y;*
- *for $2^{-54} \leq |x| < 2^{-30}$, if $\epsilon \leq 2^{-53-104-1} = 2^{-158}$ then rounding y^* is equivalent to rounding y;*

The case $|x| < 2^{-54}$ is easily dealt with using Tables 10.3 and 10.4.

[9] 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 mantissa 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 mantissa distance is less than $2\epsilon_r$.

This function	can be replaced by	when
$\exp(\epsilon), \epsilon \geq 0$	1	$\epsilon < 2^{-52}$
$\exp(\epsilon), \epsilon < 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert \leq 2^{-53}$
$\ln(1+\epsilon), \epsilon \neq 0$	ϵ^-	$\lvert\epsilon\rvert < \sqrt{2} \times 2^{-52}$
$2^\epsilon, \epsilon \geq 0$	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^-	$\epsilon \leq 1.817\cdots \times 2^{-26}$
$\sin(\epsilon), \sinh^{-1}(\epsilon), \epsilon \leq 0$	ϵ	$\lvert\epsilon\rvert \leq 1.817\cdots \times 2^{-26}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon \geq 0$	ϵ	$\epsilon \leq 1.817\cdots \times 2^{-26}$
$\arcsin(\epsilon), \sinh(\epsilon), \epsilon < 0$	ϵ^-	$\epsilon \leq 1.817\cdots \times 2^{-26}$
$\cos(\epsilon), \epsilon \neq 0$	$1^- = 1 - 2^{-53}$	$\lvert\epsilon\rvert < 2^{-26}$
$\cosh(\epsilon), \epsilon \neq 0$	1	$\lvert\epsilon\rvert < \sqrt{2} \times 2^{-26}$
$\tan(\epsilon), \tanh^{-1}(\epsilon), \epsilon \geq 0$	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	$\lvert\epsilon\rvert \leq 1.4422\cdots \times 2^{-26}$

Table 10.4: *Some results for small values in double-precision, assuming* ***rounding towards*** $-\infty$*. These results make finding worst cases useless for negative exponents of large absolute value.* x^- *is the largest FP number strictly less than* x*.*

Theorem 17 (Computation of logarithms) *Let* y *be the natural (radix-*e*) logarithm of a double-precision number* x*. Let* y^* *be an approximation to* y *such that the mantissa 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*.*

Stehlé, Lefèvre and Zimmermann are working on methods that could, one day, allow us to get the worst cases for precisions higher than double-precision [292]. They have already obtained the worst cases for function 2^x in double-extended precision, with $1/2 \leq x \leq 1$.

Conclusion

We have shown that correct rounding of the elementary functions, at least in some domains, is possible. It may seem expensive, 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 slightly more

Interval	worst case (binary)
$[\ln(2^{-1074}), -2^{-30}]$	$\exp(-1.11101101001100011000111011111011011000100111111101010 \times 2^{-27})$ $= 1.11111111111111111111111111000010010110011100111000100 \quad 1 \quad 1^{59}0001... \times 2^{-1}$
$[-2^{-30}, \ln(1-2^{-46})]$	$\exp(-1.01000000000000000000000000000000000000000000000000110010 \times 2^{-46})$ $= 1.11111111111111111111111111111111111111111111101100000 \quad 0 \quad 0^{84}1010... \times 2^{-1}$
$[\ln(1-2^{-46}), -2^{-53})$	$\exp(-1.00000000000000000000000000000000000000000000000000000001 \times 2^{-51})$ $= 1.11111111111111111111111111111111111111111111111111100 \quad 0 \quad 0^{100}1010... \times 2^{-1}$
$(2^{-52}, \ln(1+2^{-45})]$	$\exp(1.11111111111111111111111111111111111111111111111111111111 \times 2^{-53})$ $= 1.00000000000000000000000000000000000000000000000000000 \quad 1 \quad 1^{104}0101...$
$[\ln(1+2^{-45}), 2^{-30}]$	$\exp(1.11111111111111111111111111111111111111111111111110000000 \times 2^{-46})$ $= 1.00000000000000000000000000000000000000000000001111111 \quad 1 \quad 1^{83}0101...$
	$\exp(1.00011111111111111111111111111111111111111111111110101111 \times 2^{-45})$ $= 1.00000000000000000000000000000000000000000000010001111 \quad 1 \quad 1^{83}0000...$
$[2^{-30}, \ln(2^{1024})]$	$\exp(110.00001111010100101111001101111010111011001111110100)$ $= 110101100.01010000101101000000100111001000101011101110 \quad 0 \quad 0^{57}1000...$

Table 10.5: *Worst cases for the exponential function in the full double-precision range (the input values between -2^{-53} and 2^{-52} are so small that the results given in Tables 10.3 and 10.4 can be applied, so they are omitted here) [208]. Exponentials of numbers less than $\ln(2^{-1074})$ are underflows. Exponentials of numbers larger than $\ln(2^{1024})$ are overflows. 1^{59} means "a chain of 59 consecutive ones."*

Interval	worst case (binary)
$[2^{-1074}, 1)$	$\log(1.1110101001110001110110000101110011101110000000100000 \times 2^{-509})$
	$= -101100000.00101001011010100110011010110100001011111111 \quad 1 \quad 1^{60}0000...$
	$\log(1.1001010001110110111000110000010011001101011111000111 \times 2^{-384})$
	$= -100001001.10110110000011001010111101000111101100110101 \quad 1 \quad 0^{60}1010...$
	$\log(1.0010011011101001110001001101001100100111100101100000 \times 2^{-232})$
	$= -10100000.10101011001011000010010111100110100001000010 \quad 0 \quad 0^{60}1001...$
	$\log(1.0110000100111001010101011101110010000000001011111000 \times 2^{-35})$
	$= -10111.11110000001011111001101110101111011000000110101 \quad 0 \quad 1^{60}0011...$
$(1, 2^{1024}]$	$\log(1.0110001010101000100001100001001101100010100110110110 \times 2^{678})$
	$= 111010110.01000111100111101011101001111100100101110001 \quad 0 \quad 0^{64}1110...$

Table 10.6: *Worst cases for the natural (radix e) logarithm in the full double-precision range [208].*

Interval	worst case (binary)
$[-1074, 0)$	$2 ** (-1.0010100001100011101010111010111010101111011110110010 \times 2^{-15})$ $= 0.1111111111111110011001010001111101000110000011110 1111 \quad 0 \quad 0^{57}1110...$
	$2 ** (-1.0100000101101111011011000110010001000101101011001111 \times 2^{-20})$ $= 0.11111111111111111111001000010011001010111010011001110 \quad 1 \quad 1^{57}0000...$
	$2 ** (-1.0000010101010110000000011100100010101011001111110001 \times 2^{-32})$ $= 0.11111111111111111111111111111111110100101011011011 00001 \quad 1 \quad 1^{57}0000...$
	$2 ** (-1.0001100001011011100011011011011011010101100000011101 \times 2^{-33})$ $= 0.111111111111111111111111111111111111001111011010101111 00 \quad 0 \quad 0^{57}1100...$
$(0, 1024]$	$2 ** (1.1011111110111011110111100100010011101101111111000101 \times 2^{-25})$ $= 1.0000000000000000000000010011011001011000011100001 01 \quad 0 \quad 0^{59}1011...$
	$2 ** (1.1110010001011001011001010010011010111111100101001101 \times 2^{-10})$ $= 1.00000000010100111111110000101110110000101011010100 11 \quad 0 \quad 1^{59}0100...$

Table 10.7: *Worst cases for the radix-2 exponential function 2^x in the full double-precision range. Integral values of x are omitted [208].*

Interval	worst case (binary)
$(0, 1/2)$	$\log_2(1.0110000101010101010111110111010110001000010110110100 \times 2^{-513})$ $= -1000000000.10001000111111010010111111000010110010001100 \quad 0 \quad 0^{55}1100...$
$(1/2, 2^{1024})$	$\log_2(1.0110000101010101010111110111010110001000010110110100 \times 2^{512})$ $= 1000000000.01110111000000101101000000111110100110111001 \quad 1 \quad 1^{55}0011...$

Table 10.8: *Worst cases for* $\log_2(x)$ *in the full double-precision range. Values of* x *that are integer powers of 2 are omitted. For values larger than* $1/2$*, we only give one of the worst cases: the one with exponent* 512*. The other ones have the same mantissa, and exponents between* 513 *and* 1023*. For values smaller than* $1/2$*, we also give one of the worst cases only: the one with exponent* -513*. The other ones have the same mantissa and exponents between* -1024 *and* -514 *[208].*

Interval	worst case (binary)
$[2^{-24}, 2^{-17}]$	$\sin 1.11100000000000000000000000000000000000000000111000010 \times 2^{-20}$ $= 1.11011111111111111111111111111111111111111111000000101110 \quad 0 \quad 0^{72}1110... \times 2^{-20}$
$[2^{-17}, 1/32]$	$\sin 1.01011001100010111010111010011110011000110010111101110 \times 2^{-7}$ $= 1.01011001100010100000101011101010010010001001100101110 \quad 0 \quad 1^{59}0000... \times 2^{-7}$
$[1/32, 1]$	$\sin 1.11111110011101100111011100111001110100001111011011101 \times 2^{-2}$ $= 1.11101001100101010000011100110000110001000110100101011 \quad 1 \quad 1^{65}0000... \times 2^{-2}$
$[1, 2 + \frac{4675}{8192}]$	$\sin 1.10010010000111111011010101000100010000101101000110000$ $= 0.11111111111111111111111111111111111111111111111111111111 \quad 1 \quad 1^{54}0110...$ $\sin 1.01100111010101100111010001010111011100001010010100011 \times 2^{1}$ $= 1.01001111111100110101000011100100000100101000000100001 \quad 0 \quad 0^{54}1010... \times 2^{-2}$

Table 10.9: *Worst cases for* $\sin(x)$ *in double-precision in the range* $[2^{-24}, 2 + \frac{4675}{8192}]$.

Interval	worst case (binary)
$[\sin(2^{-24}), \sin(2^{-19})]$	$\arcsin 1.11011111111111111111111111111111111111111111000000101110 \times 2^{-20}$ $= 1.11100000000000000000000000000000000000000000111000001 \quad 1 \quad 1^{72}0001... \times 2^{-20}$
$[\sin(2^{-19}), \sin(2^{-18})]$	$\arcsin 1.11011111111111111111111111111111111111111000000101110 00 \times 2^{-19}$ $= 1.11100000000000000000000000000000000000000011100000111 \quad 1 \quad 1^{66}0001... \times 2^{-19}$
$[\sin(2^{-18}), 1]$	$\arcsin 1.11101001100101010000011100110000110001000110100101 10 \times 2^{-2}$ $= 1.11111110011101100111011100111001110100001111011011 01 \quad 0 \quad 0^{64}1000... \times 2^{-2}$

Table 10.10: *Worst cases for the arc-sine function in double-precision in the range* $[\sin(2^{-24}), 1]$.

Interval	worst case (binary)
$[2^{-25}, 2^{-22}]$	$\cos 1.10000000000000000000000000000000000000000000000001001 \times 2^{-23}$ $= 0.11111111111111111111111111111111111111111111101110000 \quad 0 \quad 0^{88}1101...$
$[2^{-22}, 2^{-18}]$	$\cos 1.10000000000000000000000000000000000000000000000100100 \times 2^{-22}$ $= 0.11111111111111111111111111111111111111111110111000000 \quad 0 \quad 0^{82}1101...$
$[2^{-18}, 2^{-17}]$	$\cos 1.00100000000000000000000000000000000000000111100110000 \times 2^{-18}$ $= 0.11111111111111111111111111111111111101011110000000000 \quad 0 \quad 0^{60}1001...$
$[2^{-17}, 1/256]$	$\cos 1.00000110101101010000010101010101000011100110101100010 \times 2^{-9}$ $= 0.11111111111111111101111001001101000111111101111111110 \quad 0 \quad 0^{58}1100...$
$[1/256, 1]$	$\cos 1.10010111110011001101001111010010110001000011100011111 \times 2^{-6}$ $= 0.11111111111010111011001101011101010000111000010101000 \quad 1 \quad 1^{55}0111...$
$[1, \frac{12867}{8192}]$	$\cos(1.01101011100010100110001001110011110101111100000100001)$ $= 1.00110011011111111100010110110000011100101100010100 10 \quad 1 \quad 0^{54}1011... \times 2^{-3}$

Table 10.11: *Worst cases for* $\cos(x)$ *in double-precision in the range* $[2^{-25}, 12867/8192]$. $12867/8192$ *is slightly less than* $\pi/2$. *The input values less than* 2^{-25} *are easily handled.*

Interval	worst case (binary)
$[\cos\left(\frac{12867}{8192}\right), 1-2^{-53}]$	$\arccos(1.11111101011100110111101111101001000101000101011110 00 \times 2^{-11})$ $= 1.10010001111000000000011011010100000111011000110110 00 \quad 1 \quad 1^{62}0010...$

Table 10.12: *Worst case for* $\arccos(x)$ *in double-precision in* $[\cos(12867/8192), 1-2^{-53}] \approx [0.0001176, 1-2^{-53}]$. *It must be noticed that* $1-2^{-53}$ *is the largest FP number less than* 1.

Interval	worst case (binary)
$[2^{-25}, 2^{-18}]$	$\tan 1.1101111111111111111111111111111111111111111100011111 \times 2^{-22}$ $= 1.1110000000000000000000000000000000000000000101010001 \quad 0 \quad 1^{78}0100...2^{-22}$
$[2^{-18}, 2^{-17}]$	$\tan 1.0110011111111111111111111111111111111010000100010100 \times 2^{-18}$ $= 1.0110100000000000000000000000000000001000111001100001 \quad 1 \quad 1^{57}0100...2^{-18}$
$[2^{-17}, \arctan\left(\frac{1}{2}\right)]$	$\tan(1.0101000000100100001101011001011111000001110000000010100 \times 2^{-5})$ $= 1.0101000001111000110011101011111111111001110001110010 \quad 1 \quad 0^{57}1001... \times 2^{-5}$
$[\arctan\left(\frac{1}{2}\right), \arctan(2)]$	$\tan(0.1010001101010110000110111001000100100001101010011011 0)$ $= 0.1011110111010010010011111011100111001100000101001111 0 \quad 1 \quad 1^{54}0011...$

Table 10.13: *Worst cases for* $\tan(x)$ *in double-precision in* $[2^{-25}, \arctan(2)]$, *with* $\arctan(2) \approx 1.107148$.

Interval	worst case (binary)
$[\tan(2^{-25}), \tan(2^{-18})]$	$\arctan 1.1110000000000000000000000000000000000000010101000110 \times 2^{-21}$ $= 1.1101111111111111111111111111111111111111110001111100 \quad 0 \quad 0^{72}1011... \times 2^{-21}$
$[\tan(2^{-18}), \tan\left(\frac{1}{128}\right)]$	$\arctan 1.0010001011101000110101110101111000101011110001111111 \times 2^{-11}$ $= 1.0010001011101000110101010110100101001010110100101011 \quad 1 \quad 0^{59}1101... \times 2^{-11}$
$[\tan\left(\frac{1}{128}\right), \frac{1}{2}]$	$\arctan(1.10101001001100111111111100001011101101011001101110101 \times 2^{-3})$ $= 1.10100011001111111001100101010110001011100111010110100 \quad 1 \quad 1^{55}0110... \times 2^{-3}$
$[\frac{1}{2}, 2]$	$\arctan(0.10111101110100100100111110111001110011000001010011111)$ $= 0.10100011010101100001101110010001001000011010100110110 \quad 0 \quad 0^{55}1111...$

Table 10.14: *Worst cases for* $\arctan(x)$ *in double-precision in* $[\tan(2^{-25}), 2]$.

than n digits, that is, the time already needed by current machines to compute an elementary function without guaranteed correct rounding. Indeed, the new SUN's Libmcr[10] and LIP-Arenaire's Crlibm[11] libraries provide correct rounding and have good performances. Research is active in this area.

Due to the knowledge of the worst cases, computing correctly rounded elementary functions will become easier during the next few years. It is therefore high time to think about a *standard*. 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 ?
- if "cheaper" rounding modes are provided, should a flag be raised when the result is not correctly rounded ?
- providing exactly rounded functions in the entire 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 [96]. Defour, Hanrot, Lefèvre, Muller, Revol and Zimmermann [103] recently presented some aspects of what a standard for the implementation of the mathematical functions could be.

[10] See `http://www.sun.com/download/products.xml?id=41797765`.
[11] See `https://lipforge.ens-lyon.fr/projects/crlibm/`.

Chapter 11

Miscellaneous

11.1 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 double-precision 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,

[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 [221] 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 $\sqrt{x}$).

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 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 9, where we saw (Table 9.3) that the IEEE double-precision 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 double-precision format.[3] The same argument shows that the sine or cosine of a normalized double precision floating-point number cannot be a subnormal number. However, this does not mean that this problem will never occur in another floating-point format.

11.1.1 NaNs

Each time $f(x)$ is not mathematically defined and cannot be 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 [182], 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.

11.1.2 Exact results

It is difficult to know when functions such as x^y give an exact result. However, using a theorem due to Lindemann,[5] one can show that the sine, cosine, tangent,

[3] And it never occurs in the IEEE single-precision format either.

[4] An exception is 0^0. There seems to be a general consensus to return 1 (mainly because some mathematical relations that are true for all the integers but 0 remain true for 0 with this convention; this is discussed in [161]), 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 [152] 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

$$\lim_{x\to 0} f(x)^{g(x)} = 1.$$

[5] That theorem, already used in the previous chapter, shows that if x is a nonzero algebraic number, then $\exp(x)$ is transcendental.

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. 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 (assuming a radix-2 representation):
  1. $\log_2(0) = -\infty$;
  2. $\log_2(+\infty) = +\infty$;
  3. $\log_2(1) = 0$;
  4. for any integer p such that 2^p is exactly representable (i.e., p is between the smallest possible exponent — unless denormal numbers are allowed — and the largest one), $\log_2(1.0 \times 2^p) = p$;
  5. $2^0 = 1$;
  6. $2^{-\infty} = 0$;
  7. $2^{+\infty} = +\infty$;
  8. for any integer p between the smallest possible exponent (E_{min}) and the largest one, $2^p = 1.0 \times 2^p$. If denormal numbers are allowed, values of p between E_{min} minus the number of mantissa bits and E_{min} must be added ;
- for the sine, cosine, tangent, and arctangent functions:
  1. $\sin(0) = 0$;
  2. $\cos(0) = 1$;
  3. $\tan(0) = 0$;
  4. $\arctan(0) = 0$;

[6] Gal and Bachelis [148] 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.

- for the hyperbolic sine, cosine, tangent, and arctangent functions:
  1. $\sinh(0) = 0$;
  2. $\sinh(-\infty) = -\infty$;
  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. $\tanh^{-1}(1) = +\infty$.

11.2 Notes on x^y

The power function $f(x, y) = x^y$ is very difficult to implement if we want good accuracy [64, 287]. A straightforward use of the formula

$$x^y = \exp(y \ln(x))$$

is to be avoided. 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 p of $y \ln(x)$ satisfies:

$$p = (1 + \epsilon) y \ln(x).$$

If we neglect the error due to the implementation of the exponential function itself, we find that the computed value of x^y will be:

$$\begin{aligned} & \exp(p) \\ = & \exp(y \ln(x) + y\epsilon \ln(x)) \\ = & x^y \times e^{\epsilon y \ln(x)}. \end{aligned}$$

[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.

x^y	Error in ulps
$(113/64)^{2745497944039423/2199023255552}$	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
$3482^{3062188649005575/35184372088832}$	1200.13

Table 11.1: *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 exactly rounded to the nearest, in IEEE-754 double-precision arithmetic. The worst case found during our experimentations was* 1200.13 *ulps for* $3482^{3062188649005575/35184372088832}$*, but it is very likely that there are worse cases.*

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 10 for a discussion on that problem), the error could be large. Table 11.1 gives the error obtained by computing x^y as $\exp(y\ln(x))$ in IEEE-754 double-precision 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,

$$x^y \approx 2^{Expmax+1},$$

which gives $y\ln(x) = (Expmax+1)\ln(2)$. 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

$$\left|e^{y\epsilon\ln(x)} - 1\right| < \alpha,$$

[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 $\exp(y\ln(x))$.

which is approximately equivalent to

$$y\epsilon \ln(x) < \alpha.$$

This gives

$$\epsilon < \frac{\alpha}{(Expmax + 1)\ln(2)}.$$

Therefore we must have

$$-\log_2(\epsilon) > -\log_2(\alpha) + \log_2(Expmax + 1) + \log_2(\ln(2)).$$

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 towards high quality arithmetic environments, such a requirement seems difficult to fulfill for the power function if speed is at stake. The SUN LIBMCR (see Section 12.5) provides a correctly rounded power function in double precision, 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, that 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.

11.3 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 [67], and later wrote a more portable package, SPECFUN [71]. 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 [222]. Cody also worked on the performance evaluation of programs for these functions [70, 75].

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 [21], and a discussion on the definition of these functions can be found in reference [82]. 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 [180] brings this problem to the fore and gives elegant solutions. Hull, Fairgrieve and Tang give reliable and accurate algorithms for the common complex elementary functions [172, 173]. Cody wrote a package named CELEFUNT [72] for testing a complex elementary function library.

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 [60], Higham and Smith give an algorithm for computing cosines of matrices [166], and Iserles and Zanna recently published a paper on the computation of matrix exponentials [177].

Chapter 12

Examples of Implementation

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. Some of these implementations are rather old. My purpose, here, is to show, through some examples, how the various techniques described in this book can be used.

12.1 Example 1: The Cyrix FastMath Processor

In the Cyrix Fastmath processor [86, 140], five "core" functions are directly implemented: sine, tangent, arctangent, $2^x - 1$, and $\log_2(x+1)$. The other functions are implemented using the core functions. The approximations used are shown to be monotonic [141]. For example:

- to compute $2^x - 1$ on $[-1, 1]$, a rational approximation of $2^{x/2} - 1$ is first computed, and then the identity

  $$2^x - 1 = \left(2^{x/2} - 1\right)\left[\left(2^{x/2} - 1\right) + 2\right]$$

  is used;

- to compute $\sin(x)$ on $[-\pi/4, +\pi/4]$, an odd polynomial approximation of the form $xP(x^2)$ is used. The cosine is obtained as

  $$\cos(x) = \pm\sqrt{1 - \sin^2(x)}.$$

  On the Cyrix FastMath processor, the square root is almost as fast as division, so there is little penalty from using it;

- to compute $\log_2(x + 1)$ on $[1/\sqrt{2} - 1, \sqrt{2} - 1]$, the number

  $$g = \frac{x}{x + 2}$$

is computed, so that

$$\log_2(1+x) = \log_2\left(\frac{1+g}{1-g}\right)$$

is an *odd* function of g. Then an odd rational approximation of the form $g \times Q(g^2)$ is 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 $xP(x^2)/Q(x^2)$ is used;
- to compute $\arctan(x)$ on $[-\pi/32, +\pi/32]$, a five-segment[1] odd rational approximation of the form $xP(x^2)/Q(x^2)$ is used.

All approximations have 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.

12.2 The INTEL Functions Designed for the Itanium Processor

As explained by Harrison, Kubaska, Story and Tang [159], 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 extended-precision format can be used to make double-precision programs very accurate;
- the fused multiply-add instruction (see Section 2.1.5) makes polynomial evaluation faster and in general 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.

[1] The interval is split into five sub-intervals, and a different approximation is chosen for each sub-interval. The advantages of such a method are described in the next chapter.

The designers of Intel's library for the IA-64 architecture [83, 159, 295] 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 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 — see Section 3.8.2 — can be used thanks to the available parallelism), nor in terms of accuracy when double-precision is at stake (thanks to the availability of an internal extended-precision format).

The latency of their double-precision 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 [159].

12.2.1 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 Cody and Waite's method — see Section 9.2 —, made more accurate by the availability of the fused multiply-add instruction and the internal extended-precision format;

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 Section 3.7.2;

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.

12.2.2 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

$$\arctan(x) = \text{sign }(x)\frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) \tag{12.1}$$

but since a direct use of (12.1) might sometimes lead to inaccuracies and requires a time-consuming division, some care is taken. More precisely:

1. If $|x| < 1$, the authors of [159] 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 Section 3.7.2.

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 $x^{45}\arctan(1/x)$.

12.3 The LIBULTIM Library

The LIBULTIM library (also called MathLib) was developed by Ziv and colleagues at IBM. Although it seems that 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 worst cases for correct rounding in double-precision were not known (they still are unknown for many functions and domains), the authors assumed that performing the intermediate calculations with 800 bits of precision was enough. Of course, all calculations were not performed with such a huge precision, Ziv's "multilevel strategy" [328] (see Chapter 10) 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.

12.4 The CRLIBM Library

The CRLIBM library[2] [92] was developed by the Arenaire research group in Lyon, France. It aims at returning correctly rounded results in double-precision, and uses the worst cases for rounding generated using Lefèvre's algorithms [206, 208] (see Chapter 10). The first functions of CRLIBM were written by Defour during the preparation of his PhD [101].

Ziv's multilevel strategy is used, but to guarantee correct roundings, two steps only are necessary here: the knowledge of the worst cases for correct rounding 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). In most cases this suffices to return a correctly rounded result. In the following, we call this step the *quick phase* of the algorithm;

- when the quick phase 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 phase* of the algorithm.

The authors of CRLIBM publish, with each function, a proof of its behavior. An ad-hoc multiple-precision library, called SCSLIB [102], was designed for the accurate phase. Although that may seem strange at first glance, the second phase is much simpler than the first one: first, performance is not that much an issue for the accurate phase, since that step will rarely be taken, second, the SCSLIB library provides much precision (indeed, more than what is actually needed). On the other hand, for the quick phase, performance is a primary concern, and

[2] See `http://lipforge.ens-lyon.fr/projects/crlibm/`.

tight error bounds must be obtained (using for instance methods such as the one presented in Section 3.9).

Let us now give two examples.

12.4.1 Computation of $\sin(x)$ or $\cos(x)$ (quick phase)

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

$$x = k.\pi/256 + y$$

where the reduced argument y is computed as a "double-double" $y_h + y\ell$ (i.e., the sum of two double-precision numbers) and belongs to $[-\pi/512, \pi/512]$. Notice that $\pi/512 < 2^{-7}$. 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 a small polynomial, two double-precision numbers, e and f, defined by $\cos(y) = 1 + e$ and $\sin(y) = (y_h + y\ell)(1 + f)$. This gives 14 extra bits of accuracy, so this first step is very accurate.

12.4.2 Computation of $\ln(x)$

The quick phase is accurate to 57 or 64 bits, depending on the execution path. The accurate phase is accurate to 120 bits, which suffices to guarantee correct rounding (see Table 10.6).

Subnormal input numbers are handled using

$$\ln(x) = -52\ln(2) + \ln\left(2^{52}x\right).$$

Now, if x is a normalized positive floating-point number, range reduction is very simple. First, the mantissa of x is divided by 2 if needed, so that we can write

$$x = y2^E$$

with

$$\frac{11}{16} < y < \frac{23}{16}.$$

The interval $[11/16, 23/16]$ is split into 8 subintervals. The final range reduction is done by subtracting from y the middle value of the subinterval where it lies, to get a new value z. This is done without error (see Theorem 2, in Chapter 2).

$\ln(y)$ is then approximated by a degree-12 polynomial function of z (the coefficients of the polynomial depend on the subinterval where y lies), whose first two coefficients are exactly representable as the sum of two double-precision numbers.

12.5 SUN's LIBMCR Library

The LIBMCR library[3] was developed by K.C. Ng, N. Toda and others at SUN Microsystems. The first beta version was published in December 2004, soon before the writing of this edition, so I have not yet much information on this library. It provides correctly rounded functions in double-precision.

As an example, let us show how natural logarithms are evaluated by LIBMCR:

- first, the exceptional cases are processed ($\log(1) = 0$ and $\log(+\infty) = +\infty$, $\log(0) = -\infty$, $\log(\text{negative}) = \text{NaN}$);
- then, an "almost correctly rounded" function is called;
- if the result is too close to the middle of two consecutive floating-point numbers,[4] 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 $\log(x)$. 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 $\log(y)$ is very close to a double-precision number (at a distance less than 2^{-24} ulps). This is Gal's accurate-table method, described in Section 4.3. We therefore have

$$\log(x) = n\log(2) + \log(y) + \log(r/y).$$

Define $s = (r - y)/(r + y)$. $|s|$ is less than 0.01575, and $\log(r/y)$ is computed using the Taylor approximation

$$\log\left(\frac{r}{y}\right) = \log\left(\frac{1+s}{1-s}\right) = 2s + \frac{2}{3}s^3 + \frac{2}{5}s^5 + \cdots + \frac{2}{13}s^{13}.$$

Much care is taken for evaluating the series with 77-bit accuracy.

12.6 The HP-UX Compiler for the Itanium Processor

As noticed by Markstein [225], 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,

[3] See `http://www.sun.com/download/products.xml?id=41797765`.

[4] Assuming rounding to the nearest.

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 [214] has implemented this methodology. The library does not provide 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 double-precision is 0.502 ulps [214]).

Markstein's algorithms for elementary functions on the Itanium are presented in his excellent book [224]. 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

$$2^{2^{-L}n}$$

read from a table. A typical value of L is 10. The number 2^u is approximated by a polynomial.

Bibliography

[1] M. Abramowitz and 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, and B. Tuckerman. New scalar and vector elementary functions for the IBM system/370. *IBM Journal of Research and Development*, 30(2):126–144, March 1986.

[3] H. M. Ahmed, J. M. Delosme, and M. Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *Computer*, 15(1): 65–82, January 1982.

[4] H. Alt. Comparison of arithmetic functions with respect to Boolean circuits. In *Proceedings of the 16th ACM STOC*, pages 466–470, 1984.

[5] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[6] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50. ACM Press, New York, NY, 1991.

[7] M. Andrews and T. Mraz. Unified elementary function generator. *Microprocessors and Microsystems*, 2(5):270–274, October 1978.

[8] E. Antelo, J. D. Bruguera, J. Villalba, and E. Zapata. Redundant CORDIC rotator based on parallel prediction. In Knowles and McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 172–179. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[9] E. Antelo, T. Lang, and J. D. Bruguera. Very-high radix CORDIC rotation based on selection by rounding. *Journal of VLSI Signal Processing Systems*, 25(2):141–154, June 2000.

[10] H. M. Aus and G. A. Korn. Table-lookup/interpolation function generation for fixed-point digital computations. *IEEE Transactions on Computers*, C-18(8):745–749, August 1969.

[11] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on electronic computers*, 10:389–400, 1961. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[12] D. Bailey. Some background on kanada's recent pi calculation. Technical report, Lawrence Berkeley National Laboratory, 2003. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/index.html`.

[13] D. H. Bailey. Algorithm 719, multiprecision translation and execution of FORTRAN programs. *ACM Transactions on Mathematical Software*, 19(3): 288–319, September 1993.

[14] D. H. Bailey, J. M. Borwein, P. B. Borwein, and S. Plouffe. The quest for pi. *Mathematical Intelligencer*, 19(1):50–57, 1997.

[15] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: an arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf`.

[16] B. Le Bailly and J. P. Thiran. Computing complex polynomial chebyshev approximants on the unit circle by the real remez algorithm. *SIAM Journal on Numerical Analysis*, 36(6):1858–1877, 1999.

[17] J. C. Bajard, J. Duprat, S. Kla, and J. M. Muller. Some operators for online radix 2 computations. *Journal of Parallel and Distributed Computing*, 22(2):336–345, August 1994.

[18] J. C. Bajard, S. Kla, and J. M. Muller. BKM: A new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, 43(8): 955–963, August 1994.

[19] G. A. Baker. *Essentials of Padé Approximants*. Academic Press, New York, NY, 1975.

[20] G. A. Baker. *Padé Approximants*. Number 59 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, New York, NY, 1996.

[21] H. G. Baker. Less complex elementary functions. *ACM SIGPLAN Notices*, 27(11):15–16, 1992.

[22] P. W. Baker. Suggestion for a fast binary sine/cosine generator. *IEEE Transactions on Computers*, C-25(11), November 1976.

[23] G. Bandera, M. Gonzalez, J. Villalba, J. Hormigo, and E. L. Zapata. Evaluation of elementary functions using multimedia features. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[24] R. Barrio. A unified rounding error bound for polynomial evaluation. *Advances in Computational Mathematics*, 19(4):385–400, 2003.

[25] P. Beame, S. Cook, and H. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[26] M. Bekooij, J. Huisken, and K. Nowak. Numerical accuracy of fast fourier transforms with CORDIC arithmetic. *Journal of VLSI Signal Processing Systems*, 25(2):187–193, June 2000.

[27] J.-P. Berrut and H. D. Mittelmann. Adaptive point shifts in rational approximation with optimized denominator. *Journal of Computational and Applied Mathematics*, 164–165:81–92, 2004.

[28] C. M. Black, R. P. Burton, and T. H. Miller. The need for an industry standard of accuracy for elementary-function programs. *ACM Transactions on Mathematical Software*, 10(4):361–366, December 1984.

[29] G. Bohlender, W. Krämer, and W. L. Miranker. Grading of basic arithmetical operations and functions. Technical Report RC 19593 (86059), IBM Research Division, T. J. Watson Research Center, 1994.

[30] S. Boldo, M. Daumas, and L. Théry. Formal proofs and computations in finite precision arithmetic. In Hardin and Rioboo, editors, *Proceedings of the 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2003. Available at `http://ftp.lip6.fr/lip6/reports/2003/lip6.2003.010.pdf`.

[31] S. Boldo and J.-M. Muller. Some functions computable with a fused-mac. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[32] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[33] J. Borwein and D. Bailey. *Mathematics by Experiment: Plausible Reasoning in the 21st Century*. A. K. Peters, Natick, MA, 2004.

[34] J. M. Borwein and P. B. Borwein. The arithmetic-geometric mean and fast computation of elementary functions. *SIAM Review*, 26(3):351–366, July 1984.

[35] J. M. Borwein and P. B. Borwein. On the complexity of familiar functions and numbers. *SIAM Review*, 30(4):589–601, December 1988.

[36] P. Borwein and T. Erdélyi. *Polynomials and Polynomials Inequalities*. Graduate Texts in Mathematics, Vol. 161. Springer-Verlag, New York, NY, 1995.

[37] E. L. Braun. *Digital computer design*. Academic Press, New York, NY, 1963.

[38] K. Braune. Standard functions for real and complex point and interval arguments with dynamic accuracy. *Computing, Suppl.*, 6:159–184, 1988.

[39] R. P. Brent. On the precision attainable with various floating point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.

[40] R. P. Brent. Multiple precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Analytic Computational Complexity*. Academic Press, New York, NY, 1975.

[41] R. P. Brent. Fast multiple precision evaluation of elementary functions. *Journal of the ACM*, 23:242–251, 1976.

[42] R. P. Brent. Algorithm 524, mp, a fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):71–81, 1978.

[43] R. P. Brent. A fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.

[44] R. P. Brent. Unrestricted algorithms for elementary and special functions. In S. H. Lavington, editor, *Information Processing 80*, pages 613–619. North-Holland, Amsterdam, 1980.

[45] W. S. Briggs and D. W. Matula. A 17 x 69 bit multiply and add unit with redundant binary feedback and single cycle latency. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 163–171. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[46] N. Brisebarre, D. Defour, P. Kornerup, J.-M. Muller, and N. Revol. A new range reduction algorithm. *IEEE Transactions on Computers*, 54(3):331–339, March 2005.

[47] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[48] N. Brisebarre, J.-M. Muller, and S. Raina. Accelerating correctly rounded floating-point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8):1069–1072, August 2004.

[49] N. Brisebarre, J.-M. Muller, and A. Tisserand. Computing machine-efficient polynomial approximations. Draft, LIP Laboratory, `http://perso.ens-lyon.fr/jean-michel.muller/bmt-toms.ps`, 2004.

[50] A. Bultheel, P. Gonzales-Vera, E. Hendriksen, and O. Njastad. *Orthogonal Rational Functions*, volume 5 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, New York, NY, 1999.

[51] J. W. Carr, III, A. J. Perlis, J. E. Robertson, and N. R. Scott. A visit to computation centers in the Soviet Union. *Commun. ACM*, 2(6):8–20, 1959.

[52] 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. Available at `http://gallica.bnf.fr/scripts/ConsultationTout.exe?O=N090185`.

[53] J. R. Cavallaro and N. D. Hemkumar. Efficient complex matrix transformations with CORDIC. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 122–129. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[54] J. R. Cavallaro and F. T. Luk. CORDIC arithmetic for an SVD processor. In M. J. Irwin and R. Stefanelli, editors, *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, pages 113–120. IEEE Computer Society Press, Los Alamitos, CA, 1988.

[55] J. R. Cavallaro and F. T. Luk. Floating-point CORDIC for matrix computations. In *Proceedings of the 1988 IEEE International Conference on Computer Design*, pages 40–42, 1988.

[56] L. W. Chang and S. W. Lee. Systolic arrays for the discrete Hartley transform. *IEEE Transactions on Signal Processing*, 39(11):2411–2418, November 1991.

[57] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Library Reference Manual*. Springer-Verlag, Berlin, 1991.

[58] T. C. Chen. Automatic computation of logarithms, exponentials, ratios and square roots. *IBM Journal of Research and Development*, 16:380–388, 1972.

[59] E. W. Cheney. *Introduction to Approximation Theory*. International Series in Pure and Applied Mathematics. McGraw Hill, New York, NY, 1966.

[60] S. H. Cheng, N. J. Higham, C. S. Kenney, and A. J. Laub. Approximating the logarithm of a matrix to specified accuracy. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1112–1125, 2001.

[61] C. Y. Chow and J. E. Robertson. Logical design of a redundant binary adder. In *Proceedings of the 4th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1978.

[62] C. W. Clenshaw. Rational approximations for special functions. In D. J. Evans, editor, *Software for Numerical Mathematics*. Academic Press, New York, NY, 1974.

[63] D. Cochran. Algorithms and accuracy in the HP 35. *Hewlett Packard Journal*, 23:10–11, June 1972.

[64] W. Cody and W. Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[65] W. J. Cody. A survey of practical rational and polynomial approximation of functions. *SIAM Review*, 12(3):400–423, July 1970.

[66] W. J. Cody. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.

[67] W. J. Cody. Funpack, a package of special function subroutines. Technical Memorandum 385, Argonne National Laboratory, Argonne, IL, 1981.

[68] W. J. Cody. Implementation and testing of function software. In P. C. Messina and A. Murli, editors, *Problems and Methodologies in Mathematical Software Production*, Lecture Notes in Computer Science 142. Springer-Verlag, Berlin, 1982.

[69] W. J. Cody. MACHAR: A subroutine to dynamically determine machine parameters. *ACM Transactions on Mathematical Software*, 14(4):301–311, December 1988.

[70] W. J. Cody. Performance evaluation of programs for the error and complementary error functions. *ACM Transactions on Mathematical Software*, 16(1):29–37, March 1990.

[71] W. J. Cody. Algorithm 715: SPECFUN – a portable FORTRAN package for special function routines and test drivers. *ACM Transactions on Mathematical Software*, 19(1):22–32, March 1993.

[72] W. J. Cody. CELEFUNT: A portable test package for complex elementary functions. *ACM Transactions on Mathematical Software*, 19(1):1–21, March 1993.

[73] W. J. Cody and J. T. Coonen. Algorithm 722: Functions to support the IEEE standard for binary floating-point arithmetic. *ACM Transactions on Mathematical Software*, 19(4):443–451, December 1993.

[74] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson. A proposed radix-and-word-length-independent standard for floating-point arithmetic. *IEEE MICRO*, 4(4):86–100, August 1984.

[75] W. J. Cody and L. Stoltz. The use of taylor series to test accuracy of function programs. *ACM Transactions on Mathematical Software*, 17(1): 55–63, March 1991.

[76] M. Colishaw. Decimal floating-point: Algorism for computers. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 104–111. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[77] J.-F. Collard, P. Feautrier, and T. Risset. Construction of do loops from systems of affine constraints. *Parallel Processing Letters*, 5:421–436, 1995.

[78] V. Considine. CORDIC trigonometric function generator for DSP. In *Proceedings of 1989 International Conference on Acoustics, Speech and Signal Processing*, pages 2381–2384, 1989.

[79] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[80] J. T. Coonen. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, January 1980.

[81] G. Corbaz, J. Duprat, B. Hochet, and J. M. Muller. Implementation of a VLSI polynomial evaluator for real-time applications. In *Proceedings of ASAP91*, 1991.

[82] Robert M. Corless, David J. Jeffrey, Stephen M. Watt, and James H. Davenport. "According to Abramowitz and Stegun" or arccoth needn't be uncouth. *SIGSAM Bull.*, 34(2):58–65, 2000.

[83] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-Based Systems*. Intel Press, Hillsboro, OR, 2002.

[84] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein. Correctness proofs outline for newton-raphson based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[85] M. Cosnard, A.Guyot, B. Hochet, J. M. Muller, H. Ouaouicha, P. Paul, and E. Zysman. The FELIN arithmetic coprocessor chip. In M. J. Irwin and R. Stefanelli, editors, *Proceedings of the 8th IEEE Symposium on Computer Arithmetic (Arith-8)*. IEEE Computer Society Press, Los Alamitos, CA, 1987.

[86] TX Cyrix Corporation, Richardson. *FastMath Accuracy Report*, August 1989.

[87] TX Cyrix Corporation, Richardson. *Cyrix 6x86 Processor Data Book*, 1996.

[88] L. Dadda. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356, March 1965. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[89] D. H. Daggett. Decimal-binary conversion in CORDIC. *IRE Transactions on Electronic Computers*, EC-8(3):335–339, 1959.

[90] A. Dahan-Dalmedico and J. Pfeiffer. *Histoire des Mathématiques*. Editions du Seuil, Paris, 1986. In French.

[91] D. Daney, G. Hanrot, V. Lefèvre, V. Rouillier, and P. Zimmermann. The MPFR library. `http://www.mpfr.org`.

[92] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM, a correctly rounded elementary function library. In *SPIE 48th Annual Meeting International Symposium on Optical Science and Technology*, 2003.

[93] M. Daumas, C. Mazenc, X. Merrheim, and 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*, pages 1196–1198. IMACS, Piscataway, NJ, 1994.

[94] M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3):162–175, March 1995.

[95] H. Dawid and H. Meyr. The differential CORDIC algorithm: Constant scale factor redundant implementation without correcting iterations. *IEEE Transactions on Computers*, 45(3):307–318, March 1996.

[96] F. de Dinechin and N. Gast. Towards the post-ultimate libm. Research Report 2004-47, LIP, École normale supérieure de Lyon, 2004. Available at `http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-47.pdf`.

[97] F. de Dinechin and A. Tisserand. Some improvements on multipartite table methods. In Burgess and Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH 15)*, pages 128–135. IEEE Computer Society Press, Los Alamitos, CA, June 2001.

[98] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, March 2005.

[99] C. J. de La Vallée Poussin. *L'approximation des Fonctions d'une Variable Réelle (in French)*. Gauthier-Villars, Paris, 1919.

[100] 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`, October 2002.

[101] D. Defour. *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision (in French)*. PhD thesis, Ecole Normale Supérieure de Lyon, September 2003.

[102] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, pages 29–40, 2002.

[103] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical function implementation in floating-point arithmetic. *Numerical Algorithms*, 37(1–4):367–375, 2004.

[104] D. Defour, P. Kornerup, J.-M. Muller, and N. Revol. A new range reduction algorithm. In *Proc. 35th Asilomar Conference on Signals, Systems, and Computers*, 2001.

[105] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 3 1971.

[106] J. M. Delosme. A processor for two-dimensional symmetric eigenvalue and singular value arrays. In *21st Asilomar Conference on Circuits, Systems and Computers*, pages 217–221, 1987.

[107] J. M. Delosme. Bit-level systolic algorithms for real symmetric and hermitian eigenvalue problems. *Journal of VLSI Signal Processing*, 4:69–88, 1992.

[108] B. DeLugish. *A class of algorithms for automatic evaluation of functions and computations in a digital computer*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL, 1970.

[109] E. Deprettere, P. Dewilde, and R. Udo. Pipelined CORDIC architectures for fast VLSI filtering and array processing. In *Proceedings of ICASSP'84*, pages 41.A.6.1–41.A.6.4, 1984.

[110] E. F. Deprettere and 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 P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 272–281. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[111] A. M. Despain. Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers*, C-33(5), May 1974.

[112] J. Detrey and F. de Dinechin. Second order function approximation using a single multiplication on fpgas. In *14th Intl Conference on Field-Programmable Logic and Applications*, pages 221–230. LNCS 3203, 2004.

[113] J. Van Deun and A. Bultheel. An interpolation algorithm for orthogonal rational functions. *Journal of Computational and Applied Mathematics*, 164–165:749–762, 2004.

[114] L.S. Didier and F. Rico. High radix bkm algorithm. *Numerical Algorithms*, 37(1–4):113–125, 2004.

[115] C. B. Dunham. Rational approximation with a vanishing weight function and with a fixed value at zero. *Mathematics of Computation*, 30(133):45–47, January 1976.

[116] C. B. Dunham. Choice of basis for Chebyshev approximation. *ACM Transactions on Mathematical Software*, 8(1):21–25, 1982.

[117] C. B. Dunham. Provably monotone approximations, I. *SIGNUM Newsletter*, 22:6–11, April 1987.

[118] C. B. Dunham. Provably monotone approximations, II. *SIGNUM Newsletter*, 22:30–31, July 1987.

[119] C. B. Dunham. Feasibility of "perfect" function evaluation. *SIGNUM Newsletter*, 25(4):25–26, October 1990.

[120] C. B. Dunham. Fitting approximations to the Kuki-Cody-Waite form. *International Journal of Computer Mathematics*, 31:263–265, 1990.

[121] C. B. Dunham. Provably monotone approximations, IV. Technical Report 422, Dept. of Computer Science, The University of Western Ontario, London, Canada, 1994.

[122] Charles B. Dunham. Approximation with taylor matching at the origin. *Int. J. Comput. Math.*, 80(8):1019–1024, 2003.

[123] J. Duprat and J. M. Muller. Hardwired polynomial evaluation. *Journal of Parallel and Distributed Computing*, Special Issue on Parallelism in Computer Arithmetic(5), 1988.

[124] J. Duprat and J. M. Muller. The CORDIC algorithm: New results for fast VLSI implementation. *IEEE Transactions on Computers*, 42(2):168–178, February 1993.

[125] S. W. Ellacott. On the Faber transform and efficient numerical rational approximation. *SIAM Journal of Numerical Analysis*, 20(5):989–1000, October 1983.

[126] M. Ercegovac, T. Lang, J.M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root and some elementary functions unsing small multipliers. *IEEE Transactions on Computers*, 49(7), July 2000.

[127] M. D. Ercegovac. Radix 16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6), June 1973. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[128] M. D. Ercegovac. *A general method for evaluation of functions and computation in a digital computer*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL, 1975.

[129] M. D. Ercegovac. A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7):667–680, 1977.

[130] M. D. Ercegovac. On-line arithmetic: An overview. In *SPIE, Real Time Signal Processing VII*, pages 86–93. SPIE-The International Society for Optical Engeneering, Bellingham, WA, 1984.

[131] M. D. Ercegovac and T. Lang. Fast cosine/sine implementation using on-line CORDIC. In *Proceedings of the 21st Asilomar Conference Signals, Systems, Computers*, 1987.

[132] M. D. Ercegovac and T. Lang. On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, C-36(7), July 1987. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[133] M. D. Ercegovac and T. Lang. On-line scheme for computing rotation factors. *Journal of Parallel and Distributed Computing*, Special Issue on Parallelism in Computer Arithmetic(5), 1988. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[134] M. D. Ercegovac and T. Lang. Redundant and on-line CORDIC: Application to matrix triangularization and SVD. *IEEE Transactions on Computers*, 39(6):725–740, June 1990.

[135] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, MA, 1994.

[136] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[137] M. D. Ercegovac and K. S. Trivedi. On-line algorithms for division and multiplication. *IEEE Transactions on Computers*, C-26(7):681–687, 1977. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[138] A. Feldstein and R. Goodman. Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, 23:287–297, 1976.

[139] W. Ferguson. Exact computation of a sum or difference with applications to argument reduction. In S. Knowles and W. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 216–221. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[140] W. Ferguson. Private communication. Unpublished, 1997.

[141] W. Ferguson and T. Brightman. Accurate and monotone approximations of some transcendental functions. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 237–244. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[142] C. T. Fike. Methods for evaluating polynomial approximations in function evaluation routines. *Communications of the ACM*, 10(3):175–178, 1967.

[143] B. P. Flannery, W. H. Press, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes in C, 2nd Edition*. Cambridge University Press, New York, NY, 1992.

[144] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. John Wiley, New York, NY, 2001.

[145] D. Fowler and E. Robson. Square root approximations in old babylonian mathematics: YBC 7289 in context. *Historia Mathematica*, 25:366–378, 1998.

[146] W. Fraser. A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. *Journal of the ACM*, 12(3):295–314, July 1965.

[147] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations. Lecture Notes in Computer Science*, volume 235, pages 1–16. Springer-Verlag, Berlin, 1986.

[148] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[149] W. Gautschi, G. H. Golub, and G. Opfer, editors. *Applications and Computation of Orthogonal Polynomials*. International Series of Numerical Mathematics. Birkhäuser, Basel, 1999.

[150] Walter Gautschi. *Numerical Analysis: an Introduction*. Birkhäuser, Boston, MA, 1997.

[151] W. M. Gentleman and S. B. Marovitch. More on algorithms that reveal properties of floating-point arithmetic units. *Communications of the ACM*, 17(5):276–277, May 1974.

[152] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[153] T. Granlund. The GNU multiple precision arithmetic library, release 4.1.4. Accessible electronicaly at `http://www.swox.com/gmp/gmp-man-4.1.4.pdf`, September 2004.

[154] 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*, 1995.

[155] E. R. Hansen, M. L. Patrick, and R. L. C. Wang. Polynomial evaluation with scaling. *ACM Trans. Math. Softw.*, 16(1):86–93, 1990.

[156] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi. A high-speed multiplier using a redundant binary adder tree. *IEEE Journal of Solid-State Circuits*, SC-22(1):28–34, February 1987. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[157] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, Berlin, 1999.

[158] J. Harrison. Formal verification of floating point trigonometric functions. In W.A. Hunt and S.D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000*, number 1954 in Lecture Notes in Computer Science, pages 217–233. Springer-Verlag, Berlin, 2000.

[159] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999. Available at `http://developer.intel.com/technology/itj/q41999/articles/art_5.htm`.

[160] J. F. Hart, E. W. Cheney, C. L. Lawson, H. J. Maehly, C. K. Mesztenyi, J. R. Rice, H. G. Thacher, and C. Witzgall. *Computer Approximations*. Wiley, New York, 1968.

[161] J. R. Hauser. Handling floating-point exceptions in numeric programs. Technical Report UCB//CSD-95-870, Computer Science Division, University of California, Berkeley, CA, March 1995.

[162] G. H. Haviland and A. A. Tuszinsky. A CORDIC arithmetic processor chip. *IEEE Transactions on Computers*, C-29(2), February 1980.

[163] G. H. Hekstra and E. F. A. Deprettere. Floating-point CORDIC. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 130–137. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[164] N. D. Hemkumar and J. R. Cavallaro. Redundant and on-line CORDIC for unitary transformations. *IEEE Transactions on Computers*, 43(8):941–954, August 1994.

[165] N. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, Philadelphia, PA, 2002.

[166] N. J. Higham and M. I. Smith. Computing the matrix cosine. *Numerical Algorithms*, 34:13–26, 2003.

[167] S. F. Hsiao and J. M. Delosme. Householder CORDIC algorithms. *IEEE Transactions on Computers*, 44(8):990–1000, August 1995.

[168] S.F. Hsiao, C.Y. Lau, and J.-M. Delosme. Redundant constant-factor implementation of multi-dimensional CORDIC and its application to complex SVD. *Journal of VLSI Signal Processing Systems*, 25(2):155–166, June 2000.

[169] X. Hu, S. C. Bass, and R. G. Harber. An efficient implementation of singular value decomposition rotation transformations with CORDIC processors. *Journal of Parallel and Distributed Computing*, 17:360–362, 1993.

[170] Y. H. Hu. The quantization effects of the CORDIC algorithm. *IEEE Transactions on Signal Processing*, 40(4):834–844, 1992.

[171] Y. H. Hu and S. Naganathan. An angle recoding method for CORDIC algorithm implementation. *IEEE Transactions on Computers*, 42(1):99–102, January 1993.

[172] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. Implementing complex elementary functions using exception handling. *ACM Transactions on Mathematical Software*, 20(2):215–244, June 1994.

[173] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. Implementing the complex arcsine and arccosine functions using exception handling. *ACM Transactions on Mathematical Software*, 23(3):299–335, September 1997.

[174] K. Hwang. *Computer Arithmetic Principles, Architecture and Design*. John Wiley, New York, NY, 1979.

[175] L. Imbert, J. M. Muller, and F. Rico. Radix-10 BKM algorithm for computing transcendentals on a pocket computer. *Journal of VLSI Signal Processing*, 25(2):179–186, June 2000.

[176] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE standard for radix independent floating-point arithmetic. *ANSI/IEEE Standard, Std 854-1987, New York*, 1987.

[177] A. Iserles and A. Zanna. Efficient computation of the matrix exponential by generalized polar decompositions. *SIAM Journal on Numerical Analysis*, 42(5):2218–2256, 2005.

[178] V. K. Jain and L. Lin. High-speed double precision computation of nonlinear functions. In S. Knowles and W. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 107–114. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[179] 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.

[180] W. Kahan. Branch cuts for complex elementary functions. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis*, pages 165–211. Clarendon Press, Oxford, 1987.

[181] W. Kahan. Paradoxes in concepts of accuracy. In *Lecture notes from Joint Seminar on Issues and Directions in Scientific Computations, U.C. Berkeley*, 1989.

[182] W. Kahan. Lecture notes on the status of IEEE-754. PDF file accessible electronically through the Internet at the address `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1996.

[183] W. Kahan. IEEE 754: An interview with William Kahan. *Computer*, 31(3): 114–115, March 1998.

[184] A. Karatsuba and Yu 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.

[185] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, December 1997.

[186] R. Karpinsky. PARANOIA: A floating-point benchmark. *BYTE*, 10(2), 1985.

[187] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison-Wesley, Reading, MA, 1998.

[188] D. E. Knuth. Evaluation of polynomials by computer. *Commun. ACM*, 5(12):595–599, 1962.

[189] D. König and J. F. Böhme. Optimizing the CORDIC algorithm for processors with pipeline architectures. In L. Torres, E. Masgrau, and M. A. Lagunas, editors, *Signal Processing V: Theories and Applications*. Elsevier Science, Amsterdam, 1990.

[190] I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[191] I. Koren and O. Zinaty. Evaluating elementary functions in a numerical coprocessor based on rational approximations. *IEEE Transactions on Computers*, 39(8):1030–1037, August 1990.

[192] K. Kota and J. R. Cavallaro. Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special-purpose processors. *IEEE Transactions on Computers*, 42(7):769–779, July 1993.

[193] W. Krämer. Inverse standard functions for real and complex point and interval arguments with dynamic accuracy. *Computing, Suppl.*, 6:185–212, 1988.

[194] J. Kropa. Calculator algorithms. *Mathematics Magazine*, 51(2):106–109, March 1978.

[195] H. Kuki and W. J. Cody. A statistical study of the accuracy of floating point number systems. *Communications of the ACM*, 16(14):223–230, April 1973.

[196] U. W. Kulisch. Mathematical foundation of computer arithmetic. *IEEE Transactions on Computers*, C-26(7):610–621, July 1977.

[197] U. W. Kulisch and W. L. Miranker. *Computer arithmetic in theory and practice*. Academic Press, New York, NY, 1981.

[198] T. Lang and 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, 1997.

[199] T. Lang and E. Antelo. Cordic-based computation of arccos and $\sqrt{1-t^2}$. *J. VLSI Signal Process. Syst.*, 25(1):19–38, 2000.

[200] T. Lang and J. A. Lee. SVD by constant-factor-redundant CORDIC. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 264–271. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[201] P. J. Laurent. *Approximation et Optimisation*. Enseignement des Sciences (in French). Hermann, Paris, France, 1972.

[202] D.-U Lee, W. Luk, J. Villasenor, and P. Y. K. Cheng. Non-uniform segmentation for hardware function evaluation. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, number 2778 in Lecture Notes in Computer Science, pages 796–807. Springer-Verlag, Berlin, 2003.

[203] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung. Hierarchical segmentation schemes for function evaluation. In *Proceedings of the IEEE int. Conference on Field-Programmable Technology*, pages 92–99, 2003.

[204] D.-U. Lee, O. Mencer, D. J. Pearce, and W. Luk. Automating optimized table-with-polynomial function evaluation for fpgas. In J. Becker, M. Platzner, and S. Vernalde, editors, *Proceedings of FPL 2004*, number 3203 in Lecture Notes in Computer Science, pages 364–373. Springer-Verlag, Berlin, 2004.

[205] V. Lefèvre. *Developments in Reliable Computing*, chapter An Algorithm That Computes a Lower Bound on the Distance Between a Segment and Z^2, pages 203–212. Kluwer Academic Publishers, Dordrecht, 1999.

[206] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[207] V. Lefèvre. New results on the distance between a segment and z^2. application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[208] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In Burgess and Ciminiera, editors, *Proc. of the 15th IEEE Symposium on Computer Arithmetic (Arith-15)*. IEEE Computer Society Press, Los Alamitos, CA, 2001.

[209] V. Lefèvre and J.-M. Muller. On-the-fly range reduction. *Journal of VLSI Signal Processing*, 33(1/2):31–35, February 2003.

[210] V. Lefèvre, J. M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[211] V. Lefèvre, J. M. Muller, and A. Tisserand. Toward correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.

[212] R.-C. Li. Near optimality of Chebyshev interpolation for elementary function computation. *IEEE Transactions on Computers*, 53(6):678–687, June 2004.

[213] R.-C. Li, S. Boldo, and M. Daumas. Theorems on efficient argument reduction. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, pages 129–136. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[214] R.-C. Li, P. Markstein, J. P. Okada, and J. W. 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`.

[215] A. A. Liddicoat. *High-Performance Arithmetic for Division and the Elementary Functions*. PhD thesis, Dept. of Electrical Engineering, Stanford University, Palo Alto, CA, February 2002.

[216] H. Lin and H. J. Sips. On-line CORDIC algorithms. In M. D. Ercegovac and E. Swartzlander, editors, *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 26–33. IEEE Computer Society Press, Los Alamitos, CA, 1989.

[217] H. Lin and H. J. Sips. On-line CORDIC algorithms. *IEEE Transactions on Computers*, 39(8), August 1990.

[218] R. J. Linhardt and H. S. Miller. Digit-by-digit transcendental function computation. *RCA Review*, 30:209–247, 1969. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[219] 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.

[220] W. Luther. Highly accurate tables for elementary functions. *BIT*, 35:352–360, 1995.

[221] T. Lynch and E. E. Swartzlander. A formalization for computer arithmetic. In L. Atanassova and J. Hertzberger, editors, *Computer Arithmetic and Enclosure Methods*, pages 137–145. Elsevier Science, Amsterdam, 1992.

[222] Allan J. MacLeod. Algorithm 757; miscfun, a software package to compute uncommon special functions. *ACM Trans. Math. Softw.*, 22(3):288–301, 1996.

[223] M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM*, 15(11):949–951, November 1972.

[224] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, Englewood Cliffs, NJ, 2000.

[225] P. Markstein. Accelerating sine and cosine evaluation with compiler assistance. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH16)*, pages 137–140. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[226] P. W. Markstein. Computation of elementary functions on the IBM risc system/6000 processor. *IBM Journal of Research and Development*, 34(1):111–119, January 1990.

[227] C. Mazenc, X. Merrheim, and J. M. Muller. Computing functions cos^{-1} and sin^{-1} using CORDIC. *IEEE Transactions on Computers*, 42(1):118–122, January 1993.

[228] J. E. Meggitt. Pseudo division and pseudo multiplication processes. *IBM Journal of Research and Development*, 6:210–226, 1962.

[229] X. Merrheim. *Bases discrètes et calcul des fonctions élémentaires par matériel (in French)*. PhD thesis, École Normale Supérieure de Lyon and Université Lyon I, France, February 1994.

[230] P. Midy and Y. Yakovlev. Computing some elementary functions of a complex variable. *Mathematics and Computers in Simulation*, 33:33–49, 1991.

[231] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[232] P. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, March 2005.

[233] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[234] J. M. Muller. Discrete basis and computation of elementary functions. *IEEE Transactions on Computers*, C-34(9), September 1985.

[235] J. M. Muller. *Méthodologies de calcul des fonctions élémentaires (in French)*. PhD thesis, Institut National Polytechnique de Grenoble, France, September 1985.

[236] J. M. Muller. Une méthodologie du calcul hardware des fonctions élémentaires (in French). *M2AN*, 20(4):667–695, December 1986.

[237] J.-M. Muller. On the definition of ulp(x). Technical Report 2005-09, LIP Laboratory, ENS Lyon, `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2005/RR2005-09.pdf`, 2005.

[238] J. M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3):279–288, August 1999.

[239] A. Munk-Nielsen and J. M. Muller. On-line algorithms for computing exponentials and logarithms. In *Proceedings of Europar'96, Lecture Notes in Computer Science 1124*. Springer-Verlag, Berlin, 1996.

[240] S. Nakamura. Algorithms for iterative array multiplication. *IEEE Transactions on Computers*, C-35(8), August 1986.

[241] R. Nave. Implementation of transcendental functions on a numerics processor. *Microprocessing and Microprogramming*, 11:221–225, 1983.

[242] Y. V. Nesterenko and M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers (in russian). *Mat. Zapiski*, 2:23–42, 1996.

[243] I. Newton. *Methodus Fluxionem et Serierum Infinitarum*. 1664–1671.

[244] K. C. Ng. Argument reduction for huge arguments: Good to the last bit (can be obtained by sending an e-mail to the author: kwok.ng@eng.sun.com). Technical report, SunPro, 1992.

[245] K. C. Ng and K. H. Bierman. Getting the right answer for the trigonometric functions. *SunProgrammer*, Spring 1992.

[246] S. Oberman and M. J. Flynn. Implementing division and other floating-point operations: A system perspective. In Alefeld, Fromer, and Lang, editors, *Scientific Computing and Validated Numerics (Proceedings of SCAN'95)*, pages 18–24. Akademie Verlag, Berlin, 1996.

[247] S. F. Oberman. *Design issues in high performance floating point arithmetic units*. PhD thesis, Dept. of Electrical Engineering, Stanford University, Palo Alto, CA, November 1996.

[248] S. F. Oberman. Floating-point division and square root algorithms and implementation in the AMD-k7 microprocessor. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 106–115. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[249] Y. Okabe, N. Takagi, and S. Yajima. Log-depth circuits for elementary functions using residue number system. *Electronics and Communications in Japan, Part 3*, 74:8, 1991.

[250] A. R. Omondi. *Computer Arithmetic Systems, Algorithms, Architecture and Implementations*. Prentice-Hall International Series in Computer Science, Englewood Cliffs, NJ, 1994.

[251] R. R. Osoroi, E. Antelo, J. D. Bruguera, J. Villalba, and E. Zapata. Digit on-line large radix CORDIC rotator. In P. Cappello, C. Mongenet, G. R. Perrin, P. Quinton, and Y. Robert, editors, *Proceedings of ASAP-95 (Strasbourg, France)*, pages 246–257. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[252] M. A. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, Philadelphia, PA, 2001.

[253] B. Parhami. Carry-free addition of recoded binary signed-digit numbers. *IEEE Transactions on Computers*, C-37:1470–1476, 1988.

[254] B. Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, January 1990.

[255] B. Parhami. On the implementation of arithmetic support functions for generalized signed-digit number systems. *IEEE Transactions on Computers*, 42(3):379–384, March 1993.

[256] B. Parhami. *Computer Arithmetic : Algorithms and Hardware Designs*. Oxford University Press, New-York, NY, 2000.

[257] G. Paul and M. W. Wilson. Should the elementary function library be incorporated into computer instruction sets? *ACM Transactions on Mathematical Software*, 2(2), June 1976.

[258] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[259] D. Phatak, T. Goff, and I. Koren. Constant-time addition and simultaneous format conversion based on redundant binary representations. *IEEE Transactions on Computers*, 50(11):1267–1278, November 2001.

[260] D. S. Phatak. Comments on Duprat and Muller's branching CORDIC. *IEEE Transactions on Computers*, 47(9):1037–1040, September 1998.

[261] D. S. Phatak. Double step branching CORDIC: A new algorithm for fast sine and cosine generation. *IEEE Transactions on Computers*, 47(5):587–602, May 1998.

[262] M. Pichat. Correction d'une somme en arithmétique à virgule flottante (in french). *Numerische Mathematik*, 19:400–406, 1972.

[263] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[264] C. V. Ramamoorthy, J. R. Goodman, and K. H. Kim. Some properties of iterative square-rooting methods using high-speed multiplication. *IEEE Transactions on Computers*, C-21:837–847, 1972. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[265] E. Remez. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *C.R. Académie des Sciences, Paris*, 198:2063–2065, 1934.

[266] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11:1–16, 2005.

[267] J. R. Rice. *The approximation of functions*. Addison-Wesley, Reading, MA, 1964.

[268] T. J. Rivlin. *An Introduction to the approximation of functions*. Blaisdell Publishing Company, Walham, MA, 1969. Republished by Dover, 1981.

[269] T. J. Rivlin. *Chebyshev polynomials. From approximation theory to algebra (Second edition)*. Pure and Applied Mathematics. John Wiley, New York, NY, 1990.

[270] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[271] J. E. Robertson. The correspondence between methods of digital division and multiplier recoding procedures. *IEEE Transactions on Computers*, C-19(8), August 1970.

[272] D. Rusinoff. 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 Journal of Computation and Mathematics*, 1:148–200, 1998.

[273] B. V. Sakar and E. V. Krishnamurthy. Economic pseudodivision processes for obtaining square root, logarithm and arctan. *IEEE Transactions on Computers*, C-20(12), December 1971.

[274] E. Salamin. Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, 30:565–570, 1976.

[275] D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In Knowles and McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, pages 17–28. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[276] C. W. Schelin. Calculator function approximation. *American Mathematical Monthly*, 90(5), May 1983.

[277] H. Schmid and A. Bogacki. Use decimal CORDIC for generation of many transcendental functions. *EDN*, pages 64–73, February 1973.

[278] A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7:281–292, 1971. In German.

[279] M. J. Schulte and J. Stine. Symmetric bipartite tables for accurate function approximation. In In T. Lang, J.M. Muller, and N. Takagi, editors, *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[280] M. J. Schulte and J. E. Stine. Accurate function evaluation by symmetric table lookup and addition. In Thiele, Fortes, Vissers, Taylor, Noll, and Teich, editors, *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (Zurich, Switzerland)*, pages 144–153. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[281] M. J. Schulte and E. E. Swartzlander. Exact rounding of certain elementary functions. In E. E. Swartzlander, M. J. Irwin, and G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 138–145. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[282] M. J. Schulte and E. E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8): 964–973, August 1994.

[283] M. J. Schulte and J. E. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, August 1999.

[284] P. Sebah and X. Gourdon. Newton's method and high-order iterations. Technical report, 2001. `http://numbers.computation.free.fr/Constants/Algorithms/newton.html`.

[285] A. Seznec and F. Lloansi. Étude des architectures des microproceseurs MIPS R10000, Ultrasparc et Pentium Pro (in french). Technical Report 1024, IRISA Rennes, France, May 1996.

[286] A. Seznec and T. Vauléon. Étude comparative des architectures des microprocesseurs Intel Pentium et PowerPC 601 (in french). Technical Report 835, IRISA Rennes, France, June 1994.

[287] J. D. Silverstein, S. E. Sommars, and Y. C. Tao. The UNIX system math library, a status report. In *USENIX — Winter'90*, 1990.

[288] A. Singh, D. Phatak, T. Goff, M. Riggs, J. Plusquellic, and C. Patel. Comparison of branching CORDIC implementations. In E. Deprettere, S. Bhattacharyya, J. Cavallaro, A. Darte, and L. Thiele, editors, *ASAP'03, The IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 215–225. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[289] R. A. Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995.

[290] W. H. Specker. A class of algorithms for $ln(x)$, $exp(x)$, $sin(x)$, $cos(x)$, $tan^{-1}(x)$ and $cot^{-1}(x)$. *IEEE Transactions on Electronic Computers*, EC-14, 1965. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[291] H. M. Stark. *An Introduction to Number Theory*. MIT Press, Cambridge, MA, 1981.

[292] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching worst cases of a one-variable function. *IEEE Transactions on Computers*, 54(3):340–346, March 2005.

[293] D. Stehlé and P. Zimmermann. Gal's accurate tables method revisited. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[294] J. E. Stine and M. J. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21: 167–177, 1999.

[295] S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In *Proceedings of the 14th IEEE Symposium on Computer*

Arithmetic, Adelaide, Australia, pages 4–11. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[296] D. A. Sunderland, R. A. Strauch, S. W. Wharfield, H. T. Peterson, and C. R. Cole. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid State Circuits*, sc-19(4):497–506, 1984.

[297] T. Y. Sung and Y. H. Hu. Parallel VLSI implementation of Kalman filters. *IEEE Transactions on Aerospace and Electronic Systems*, AES 23(2), March 1987.

[298] E. E. Swartzlander. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[299] E. E. Swartzlander. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[300] N. Takagi. *Studies on hardware algorithms for arithmetic operations with a redundant binary representation*. PhD thesis, Dept. Info. Sci., Kyoto University, Japan, 1987.

[301] N. Takagi, T. Asada, and S. Yajima. A hardware algorithm for computing sine and cosine using redundant binary representation. *Systems and Computers in Japan*, 18(8), 1987.

[302] N. Takagi, T. Asada, and S. Yajima. Redundant CORDIC methods with a constant scale factor. *IEEE Transactions on Computers*, 40(9):989–995, September 1991.

[303] N. Takagi, H. Yasukura, and S. Yajima. High speed multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers*, C-34(9), September 1985.

[304] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[305] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, December 1990.

[306] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[307] P. T. P. Tang. Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 18(2):211–222, June 1992.

[308] The Polylib Team. Polylib, a library of polyhedral functions, version 5.20.0. `http://icps.u-strasbg.fr/polylib/`, 2004.

[309] D. Timmermann, H. Hahn, and B. J. Hosticka. Low latency time CORDIC algorithms. *IEEE Transactions on Computers*, 41(8):1010–1015, August 1992.

[310] D. Timmermann, H. Hahn, B. J. Hosticka, and B. Rix. A new addition scheme and fast scaling factor compensation methods for CORDIC algorithms. *INTEGRATION, the VLSI Journal*, 11:85–100, 1991.

[311] D. Timmermann, H. Hahn, B. J. Hosticka, and G. Schmidt. A programmable CORDIC chip for digital signal processing applications. *IEEE Journal of Solid-State Circuits*, 26(9):1317–1321, September 1991.

[312] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.

[313] C.-Y. Tseng. A multiple-exchange algorithm for complex chebyshev approximation by polynomials on the unit circle. *SIAM Journal on Numerical Analysis*, 33(5):2017–2049, 1996.

[314] L. Veidinger. On the numerical determination of the best approximations in the Chebyshev sense. *Numerische Mathematik*, 2:99–105, 1960.

[315] B. Verdonk, A. Cuyt, and 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.

[316] B. Verdonk, A. Cuyt, and D. Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic ii: conversions. *ACM Trans. Math. Softw.*, 27(1):119–140, 2001.

[317] J. E. Volder. The CORDIC computing technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, 1959. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[318] J. E. Volder. The birth of CORDIC. *Journal of VLSI Signal Processing Systems*, 25(2):101–105, June 2000.

[319] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, pages 14–17, February 1964. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[320] P. J. L. Wallis, editor. *Improving Floating-Point Programming*. John Wiley, New York, NY, 1990.

[321] J. S. Walther. A unified algorithm for elementary functions. In *Joint Computer Conference Proceedings*, 1971. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[322] J. S. Walther. The story of unified CORDIC. *Journal of VLSI Signal Processing Systems*, 25(2):107–112, June 2000.

[323] S. Wang and E. E. Swartzlander. Merged CORDIC algorithm. In *1995 IEEE International Symposium on Circuits and Systems*, 1995.

[324] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3):278–294, March 1994.

[325] J. M. Yohe. Roundings in floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):577–586, June 1973.

[326] H. Yoshimura, T. Nakanishi, and H. Tamauchi. A 50MHz geometrical mapping processor. In *Proceedings of the 1988 IEEE International Solid-State Circuits Conference*, 1988.

[327] P. Zimmermann. Arithmétique en précision arbitraire. *Réseaux et Systèmes Répartis, Calculateurs Parallèlles*, 13(4–5):357–386, 2001. In French.

[328] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.

[329] F. Zou and P. Kornerup. High speed DCT/IDCT using a pipelined CORDIC algorithm. In Knowles and McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 180–187. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[330] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, August 1994.

Index