



Jean-Michel Muller  
Nicolas Brisebarre  
Florent de Dinechin  
Claude-Pierre Jeannerod  
Vincent Lefèvre  
Guillaume Melquiond  
Nathalie Revol  
Damien Stehlé  
Serge Torres

# Handbook of Floating-Point Arithmetic

Birkhäuser  
Boston • Basel • Berlin

Jean-Michel Muller  
CNRS, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
jean-michel.muller@  
ens-lyon.fr

Nicolas Brisebarre  
CNRS, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
nicolas.brisebarre@  
ens-lyon.fr

Florent de Dinechin  
ENSL, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
florent.de.dinechin@  
ens-lyon.fr

Claude-Pierre Jeannerod  
INRIA, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
claude-pierre.jeannerod@  
ens-lyon.fr

Vincent Lefèvre  
INRIA, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
vincent@vinc17.net

Guillaume Melquiond  
INRIA Saclay – Île-de-  
France  
Parc Orsay Université  
4, rue Jacques Monod  
91893 Orsay Cedex  
France  
guillaume.melquiond@  
inria.fr

Nathalie Revol  
INRIA, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
nathalie.revol@ens-lyon.fr

Damien Stehlé  
CNRS, Macquarie University,  
and University of Sydney  
School of Mathematics  
and Statistics  
University of Sydney  
Sydney NSW 2006  
Australia  
damien.stehle@gmail.com

Serge Torres  
ENSL, Laboratoire LIP  
École Normale  
Supérieure de Lyon  
46, allée d'Italie  
69364 Lyon Cedex 07  
France  
serge.torres@ens-lyon.fr

ISBN 978-0-8176-4704-9  
DOI 10.1007/978-0-8176-4705-6

e-ISBN 978-0-8176-4705-6

Library of Congress Control Number: 2009939668

Mathematics Subject Classification (2000): 65Y99, 68N30  
ACM Subject Classification: G.1.0, G.4

© Birkhäuser Boston, a part of Springer Science+Business Media, LLC 2010

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Birkhäuser Boston, c/o Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Birkhäuser Boston is part of Springer Science+Business Media (www.birkhauser.com)

# Contents

<b>Preface</b>	<b>xv</b>
<b>List of Figures</b>	<b>xvii</b>
<b>List of Tables</b>	<b>xxi</b>
<b>I Introduction, Basic Definitions, and Standards</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Some History . . . . .	3
1.2 Desirable Properties . . . . .	6
1.3 Some Strange Behaviors . . . . .	7
1.3.1 Some famous bugs . . . . .	7
1.3.2 Difficult problems . . . . .	8
<b>2 Definitions and Basic Notions</b>	<b>13</b>
2.1 Floating-Point Numbers . . . . .	13
2.2 Rounding . . . . .	20
2.2.1 Rounding modes . . . . .	20
2.2.2 Useful properties . . . . .	22
2.2.3 Relative error due to rounding . . . . .	23
2.3 Exceptions . . . . .	25
2.4 Lost or Preserved Properties of the Arithmetic on the Real Numbers . . . . .	27
2.5 Note on the Choice of the Radix . . . . .	29
2.5.1 Representation errors . . . . .	29
2.5.2 A case for radix 10 . . . . .	30
2.6 Tools for Manipulating Floating-Point Errors . . . . .	32
2.6.1 The ulp function . . . . .	32
2.6.2 Errors in ulps and relative errors . . . . .	37
2.6.3 An example: iterated products . . . . .	37
2.6.4 Unit roundoff . . . . .	39
2.7 Note on Radix Conversion . . . . .	40

2.7.1	Conditions on the formats . . . . .	40
2.7.2	Conversion algorithms . . . . .	43
2.8	The Fused Multiply-Add (FMA) Instruction . . . . .	51
2.9	Interval Arithmetic . . . . .	51
2.9.1	Intervals with floating-point bounds . . . . .	52
2.9.2	Optimized rounding . . . . .	52
<b>3</b>	<b>Floating-Point Formats and Environment</b>	<b>55</b>
3.1	The IEEE 754-1985 Standard . . . . .	56
3.1.1	Formats specified by IEEE 754-1985 . . . . .	56
3.1.2	Little-endian, big-endian . . . . .	60
3.1.3	Rounding modes specified by IEEE 754-1985 . . . . .	61
3.1.4	Operations specified by IEEE 754-1985 . . . . .	62
3.1.5	Exceptions specified by IEEE 754-1985 . . . . .	66
3.1.6	Special values . . . . .	69
3.2	The IEEE 854-1987 Standard . . . . .	70
3.2.1	Constraints internal to a format . . . . .	70
3.2.2	Various formats and the constraints between them . . .	71
3.2.3	Conversions between floating-point numbers and decimal strings . . . . .	72
3.2.4	Rounding . . . . .	73
3.2.5	Operations . . . . .	73
3.2.6	Comparisons . . . . .	74
3.2.7	Exceptions . . . . .	74
3.3	The Need for a Revision . . . . .	74
3.3.1	A typical problem: “double rounding” . . . . .	75
3.3.2	Various ambiguities . . . . .	77
3.4	The New IEEE 754-2008 Standard . . . . .	79
3.4.1	Formats specified by the revised standard . . . . .	80
3.4.2	Binary interchange format encodings . . . . .	81
3.4.3	Decimal interchange format encodings . . . . .	82
3.4.4	Larger formats . . . . .	92
3.4.5	Extended and extendable precisions . . . . .	92
3.4.6	Attributes . . . . .	93
3.4.7	Operations specified by the standard . . . . .	97
3.4.8	Comparisons . . . . .	99
3.4.9	Conversions . . . . .	99
3.4.10	Default exception handling . . . . .	100
3.4.11	Recommended transcendental functions . . . . .	103
3.5	Floating-Point Hardware in Current Processors . . . . .	104
3.5.1	The common hardware denominator . . . . .	104
3.5.2	Fused multiply-add . . . . .	104
3.5.3	Extended precision . . . . .	104
3.5.4	Rounding and precision control . . . . .	105

3.5.5	SIMD instructions . . . . .	106
3.5.6	Floating-point on x86 processors: SSE2 versus x87 . . .	106
3.5.7	Decimal arithmetic . . . . .	107
3.6	Floating-Point Hardware in Recent Graphics Processing Units	108
3.7	Relations with Programming Languages . . . . .	109
3.7.1	The Language Independent Arithmetic (LIA) standard	109
3.7.2	Programming languages . . . . .	110
3.8	Checking the Environment . . . . .	110
3.8.1	MACHAR . . . . .	111
3.8.2	Paranoia . . . . .	111
3.8.3	UCBTest . . . . .	115
3.8.4	TestFloat . . . . .	116
3.8.5	IeeeCC754 . . . . .	116
3.8.6	Miscellaneous . . . . .	116
<b>II Cleverly Using Floating-Point Arithmetic</b>		<b>117</b>
<b>4</b>	<b>Basic Properties and Algorithms</b>	<b>119</b>
4.1	Testing the Computational Environment . . . . .	119
4.1.1	Computing the radix . . . . .	119
4.1.2	Computing the precision . . . . .	121
4.2	Exact Operations . . . . .	122
4.2.1	Exact addition . . . . .	122
4.2.2	Exact multiplications and divisions . . . . .	124
4.3	Accurate Computations of Sums of Two Numbers . . . . .	125
4.3.1	The Fast2Sum algorithm . . . . .	126
4.3.2	The 2Sum algorithm . . . . .	129
4.3.3	If we do not use rounding to nearest . . . . .	131
4.4	Computation of Products . . . . .	132
4.4.1	Veltkamp splitting . . . . .	132
4.4.2	Dekker's multiplication algorithm . . . . .	135
4.5	Complex numbers . . . . .	139
4.5.1	Various error bounds . . . . .	140
4.5.2	Error bound for complex multiplication . . . . .	141
4.5.3	Complex division . . . . .	144
4.5.4	Complex square root . . . . .	149
<b>5</b>	<b>The Fused Multiply-Add Instruction</b>	<b>151</b>
5.1	The 2MultFMA Algorithm . . . . .	152
5.2	Computation of Residuals of Division and Square Root . . . .	153
5.3	Newton-Raphson-Based Division with an FMA . . . . .	155
5.3.1	Variants of the Newton-Raphson iteration . . . . .	155

5.3.2	Using the Newton–Raphson iteration for correctly rounded division . . . . .	160
5.4	Newton–Raphson-Based Square Root with an FMA . . . . .	167
5.4.1	The basic iterations . . . . .	167
5.4.2	Using the Newton–Raphson iteration for correctly rounded square roots . . . . .	168
5.5	Multiplication by an Arbitrary-Precision Constant . . . . .	171
5.5.1	Checking for a given constant $C$ if Algorithm 5.2 will always work . . . . .	172
5.6	Evaluation of the Error of an FMA . . . . .	175
5.7	Evaluation of Integer Powers . . . . .	177
<b>6</b>	<b>Enhanced Floating-Point Sums, Dot Products, and Polynomial Values</b>	<b>181</b>
6.1	Preliminaries . . . . .	182
6.1.1	Floating-point arithmetic models . . . . .	183
6.1.2	Notation for error analysis and classical error estimates	184
6.1.3	Properties for deriving running error bounds . . . . .	187
6.2	Computing Validated Running Error Bounds . . . . .	188
6.3	Computing Sums More Accurately . . . . .	190
6.3.1	Reordering the operands, and a bit more . . . . .	190
6.3.2	Compensated sums . . . . .	192
6.3.3	Implementing a “long accumulator” . . . . .	199
6.3.4	On the sum of three floating-point numbers . . . . .	199
6.4	Compensated Dot Products . . . . .	201
6.5	Compensated Polynomial Evaluation . . . . .	203
<b>7</b>	<b>Languages and Compilers</b>	<b>205</b>
7.1	A Play with Many Actors . . . . .	205
7.1.1	Floating-point evaluation in programming languages .	206
7.1.2	Processors, compilers, and operating systems . . . . .	208
7.1.3	In the hands of the programmer . . . . .	209
7.2	Floating Point in the C Language . . . . .	209
7.2.1	Standard C99 headers and IEEE 754-1985 support . . .	209
7.2.2	Types . . . . .	210
7.2.3	Expression evaluation . . . . .	213
7.2.4	Code transformations . . . . .	216
7.2.5	Enabling unsafe optimizations . . . . .	217
7.2.6	Summary: a few horror stories . . . . .	218
7.3	Floating-Point Arithmetic in the C++ Language . . . . .	220
7.3.1	Semantics . . . . .	220
7.3.2	Numeric limits . . . . .	221
7.3.3	Overloaded functions . . . . .	222
7.4	FORTRAN Floating Point in a Nutshell . . . . .	223

7.4.1	Philosophy . . . . .	223
7.4.2	IEEE 754 support in FORTRAN . . . . .	226
7.5	Java Floating Point in a Nutshell . . . . .	227
7.5.1	Philosophy . . . . .	227
7.5.2	Types and classes . . . . .	228
7.5.3	Infinities, NaNs, and signed zeros . . . . .	230
7.5.4	Missing features . . . . .	231
7.5.5	Reproducibility . . . . .	232
7.5.6	The BigDecimal package . . . . .	233
7.6	Conclusion . . . . .	234
<b>III Implementing Floating-Point Operators</b>		<b>237</b>
<b>8</b>	<b>Algorithms for the Five Basic Operations</b>	<b>239</b>
8.1	Overview of Basic Operation Implementation . . . . .	239
8.2	Implementing IEEE 754-2008 Rounding . . . . .	241
8.2.1	Rounding a nonzero finite value with unbounded exponent range . . . . .	241
8.2.2	Overflow . . . . .	243
8.2.3	Underflow and subnormal results . . . . .	244
8.2.4	The inexact exception . . . . .	245
8.2.5	Rounding for actual operations . . . . .	245
8.3	Floating-Point Addition and Subtraction . . . . .	246
8.3.1	Decimal addition . . . . .	249
8.3.2	Decimal addition using binary encoding . . . . .	250
8.3.3	Subnormal inputs and outputs in binary addition . . . . .	251
8.4	Floating-Point Multiplication . . . . .	251
8.4.1	Normal case . . . . .	252
8.4.2	Handling subnormal numbers in binary multiplication . . . . .	252
8.4.3	Decimal specifics . . . . .	253
8.5	Floating-Point Fused Multiply-Add . . . . .	254
8.5.1	Case analysis for normal inputs . . . . .	254
8.5.2	Handling subnormal inputs . . . . .	258
8.5.3	Handling decimal cohorts . . . . .	259
8.5.4	Overview of a binary FMA implementation . . . . .	259
8.6	Floating-Point Division . . . . .	262
8.6.1	Overview and special cases . . . . .	262
8.6.2	Computing the significand quotient . . . . .	263
8.6.3	Managing subnormal numbers . . . . .	264
8.6.4	The inexact exception . . . . .	265
8.6.5	Decimal specifics . . . . .	265
8.7	Floating-Point Square Root . . . . .	265
8.7.1	Overview and special cases . . . . .	265

8.7.2	Computing the significand square root . . . . .	266
8.7.3	Managing subnormal numbers . . . . .	267
8.7.4	The inexact exception . . . . .	267
8.7.5	Decimal specifics . . . . .	267
<b>9</b>	<b>Hardware Implementation of Floating-Point Arithmetic</b>	<b>269</b>
9.1	Introduction and Context . . . . .	269
9.1.1	Processor internal formats . . . . .	269
9.1.2	Hardware handling of subnormal numbers . . . . .	270
9.1.3	Full-custom VLSI versus reconfigurable circuits . . . . .	271
9.1.4	Hardware decimal arithmetic . . . . .	272
9.1.5	Pipelining . . . . .	273
9.2	The Primitives and Their Cost . . . . .	274
9.2.1	Integer adders . . . . .	274
9.2.2	Digit-by-integer multiplication in hardware . . . . .	280
9.2.3	Using nonstandard representations of numbers . . . . .	280
9.2.4	Binary integer multiplication . . . . .	281
9.2.5	Decimal integer multiplication . . . . .	283
9.2.6	Shifters . . . . .	284
9.2.7	Leading-zero counters . . . . .	284
9.2.8	Tables and table-based methods for fixed-point function approximation . . . . .	286
9.3	Binary Floating-Point Addition . . . . .	288
9.3.1	Overview . . . . .	288
9.3.2	A first dual-path architecture . . . . .	289
9.3.3	Leading-zero anticipation . . . . .	291
9.3.4	Probing further on floating-point adders . . . . .	295
9.4	Binary Floating-Point Multiplication . . . . .	296
9.4.1	Basic architecture . . . . .	296
9.4.2	FPGA implementation . . . . .	296
9.4.3	VLSI implementation optimized for delay . . . . .	298
9.4.4	Managing subnormals . . . . .	301
9.5	Binary Fused Multiply-Add . . . . .	302
9.5.1	Classic architecture . . . . .	303
9.5.2	To probe further . . . . .	305
9.6	Division . . . . .	305
9.6.1	Digit-recurrence division . . . . .	306
9.6.2	Decimal division . . . . .	309
9.7	Conclusion: Beyond the FPU . . . . .	309
9.7.1	Optimization in context of standard operators . . . . .	310
9.7.2	Operation with a constant operand . . . . .	311
9.7.3	Block floating point . . . . .	313
9.7.4	Specific architectures for accumulation . . . . .	313
9.7.5	Coarser-grain operators . . . . .	317

9.8	Probing Further . . . . .	320
<b>10</b>	<b>Software Implementation of Floating-Point Arithmetic</b>	<b>321</b>
10.1	Implementation Context . . . . .	322
10.1.1	Standard encoding of binary floating-point data . . . . .	322
10.1.2	Available integer operators . . . . .	323
10.1.3	First examples . . . . .	326
10.1.4	Design choices and optimizations . . . . .	328
10.2	Binary Floating-Point Addition . . . . .	329
10.2.1	Handling special values . . . . .	330
10.2.2	Computing the sign of the result . . . . .	332
10.2.3	Swapping the operands and computing the alignment shift . . . . .	333
10.2.4	Getting the correctly rounded result . . . . .	335
10.3	Binary Floating-Point Multiplication . . . . .	341
10.3.1	Handling special values . . . . .	341
10.3.2	Sign and exponent computation . . . . .	343
10.3.3	Overflow detection . . . . .	345
10.3.4	Getting the correctly rounded result . . . . .	346
10.4	Binary Floating-Point Division . . . . .	349
10.4.1	Handling special values . . . . .	350
10.4.2	Sign and exponent computation . . . . .	351
10.4.3	Overflow detection . . . . .	354
10.4.4	Getting the correctly rounded result . . . . .	355
10.5	Binary Floating-Point Square Root . . . . .	361
10.5.1	Handling special values . . . . .	362
10.5.2	Exponent computation . . . . .	364
10.5.3	Getting the correctly rounded result . . . . .	365
<b>IV</b>	<b>Elementary Functions</b>	<b>373</b>
<b>11</b>	<b>Evaluating Floating-Point Elementary Functions</b>	<b>375</b>
11.1	Basic Range Reduction Algorithms . . . . .	379
11.1.1	Cody and Waite's reduction algorithm . . . . .	379
11.1.2	Payne and Hanek's algorithm . . . . .	381
11.2	Bounding the Relative Error of Range Reduction . . . . .	382
11.3	More Sophisticated Range Reduction Algorithms . . . . .	384
11.3.1	An example of range reduction for the exponential function . . . . .	386
11.3.2	An example of range reduction for the logarithm . . . . .	387
11.4	Polynomial or Rational Approximations . . . . .	388
11.4.1	$L^2$ case . . . . .	389
11.4.2	$L^\infty$ , or minimax case . . . . .	390

11.4.3	“Truncated” approximations . . . . .	392
11.5	Evaluating Polynomials . . . . .	393
11.6	Correct Rounding of Elementary Functions to binary64 . . . .	394
11.6.1	The Table Maker’s Dilemma and Ziv’s onion peeling strategy . . . . .	394
11.6.2	When the TMD is solved . . . . .	395
11.6.3	Rounding test . . . . .	396
11.6.4	Accurate second step . . . . .	400
11.6.5	Error analysis and the accuracy/performance tradeoff	401
11.7	Computing Error Bounds . . . . .	402
11.7.1	The point with efficient code . . . . .	402
11.7.2	Example: a “double-double” polynomial evaluation . .	403
<b>12</b>	<b>Solving the Table Maker’s Dilemma</b>	<b>405</b>
12.1	Introduction . . . . .	405
12.1.1	The Table Maker’s Dilemma . . . . .	406
12.1.2	Brief history of the TMD . . . . .	410
12.1.3	Organization of the chapter . . . . .	411
12.2	Preliminary Remarks on the Table Maker’s Dilemma . . . . .	412
12.2.1	Statistical arguments: what can be expected in practice	412
12.2.2	In some domains, there is no need to find worst cases .	416
12.2.3	Deducing the worst cases from other functions or domains . . . . .	419
12.3	The Table Maker’s Dilemma for Algebraic Functions . . . . .	420
12.3.1	Algebraic and transcendental numbers and functions .	420
12.3.2	The elementary case of quotients . . . . .	422
12.3.3	Around Liouville’s theorem . . . . .	424
12.3.4	Generating bad rounding cases for the square root using Hensel 2-adic lifting . . . . .	425
12.4	Solving the Table Maker’s Dilemma for Arbitrary Functions .	429
12.4.1	Lindemann’s theorem: application to some transcendental functions . . . . .	429
12.4.2	A theorem of Nesterenko and Waldschmidt . . . . .	430
12.4.3	A first method: tabulated differences . . . . .	432
12.4.4	From the TMD to the distance between a grid and a segment . . . . .	434
12.4.5	Linear approximation: Lefèvre’s algorithm . . . . .	436
12.4.6	The SLZ algorithm . . . . .	443
12.4.7	Periodic functions on large arguments . . . . .	448
12.5	Some Results . . . . .	449
12.5.1	Worst cases for the exponential, logarithmic, trigonometric, and hyperbolic functions . . . . .	449
12.5.2	A special case: integer powers . . . . .	458
12.6	Current Limits and Perspectives . . . . .	458

<b>V Extensions</b>	<b>461</b>
<b>13 Formalisms for Certifying Floating-Point Algorithms</b>	<b>463</b>
13.1 Formalizing Floating-Point Arithmetic . . . . .	463
13.1.1 Defining floating-point numbers . . . . .	464
13.1.2 Simplifying the definition . . . . .	466
13.1.3 Defining rounding operators . . . . .	467
13.1.4 Extending the set of numbers . . . . .	470
13.2 Formalisms for Certifying Algorithms by Hand . . . . .	471
13.2.1 Hardware units . . . . .	471
13.2.2 Low-level algorithms . . . . .	472
13.2.3 Advanced algorithms . . . . .	473
13.3 Automating Proofs . . . . .	474
13.3.1 Computing on bounds . . . . .	475
13.3.2 Counting digits . . . . .	477
13.3.3 Manipulating expressions . . . . .	479
13.3.4 Handling the relative error . . . . .	483
13.4 Using Gappa . . . . .	484
13.4.1 Toy implementation of sine . . . . .	484
13.4.2 Integer division on Itanium . . . . .	488
<b>14 Extending the Precision</b>	<b>493</b>
14.1 Double-Words, Triple-Words. . . . .	494
14.1.1 Double-word arithmetic . . . . .	495
14.1.2 Static triple-word arithmetic . . . . .	498
14.1.3 Quad-word arithmetic . . . . .	500
14.2 Floating-Point Expansions . . . . .	503
14.3 Floating-Point Numbers with Batched Additional Exponent .	509
14.4 Large Precision Relying on Processor Integers . . . . .	510
14.4.1 Using arbitrary-precision integer arithmetic for arbitrary-precision floating-point arithmetic . . . . .	512
14.4.2 A brief introduction to arbitrary-precision integer arithmetic . . . . .	513
<b>VI Perspectives and Appendix</b>	<b>517</b>
<b>15 Conclusion and Perspectives</b>	<b>519</b>
<b>16 Appendix: Number Theory Tools for Floating-Point Arithmetic</b>	<b>521</b>
16.1 Continued Fractions . . . . .	521
16.2 The LLL Algorithm . . . . .	524
<b>Bibliography</b>	<b>529</b>
<b>Index</b>	<b>567</b>

# Preface

FLOATING-POINT ARITHMETIC is by far the most widely used way of approximating real-number arithmetic for performing numerical calculations on modern computers. A rough presentation of floating-point arithmetic requires only a few words: a number  $x$  is represented in radix  $\beta$  floating-point arithmetic with a sign  $s$ , a significand  $m$ , and an exponent  $e$ , such that  $x = s \times m \times \beta^e$ . Making such an arithmetic reliable, fast, and portable is however a very complex task. Although it could be argued that, to some extent, the concept of floating-point arithmetic (in radix 60) was invented by the Babylonians, or that it is the underlying arithmetic of the slide rule, its first modern implementation appeared in Konrad Zuse's 5.33Hz Z3 computer.

A vast quantity of very diverse arithmetics was implemented between the 1960s and the early 1980s. The radix (radices 2, 4, 16, and 10 were then considered), and the sizes of the significand and exponent fields were not standardized. The approaches for rounding and for handling underflows, overflows, or "forbidden operations" (such as  $5/0$  or  $\sqrt{-3}$ ) were significantly different from one machine to another. This lack of standardization made it difficult to write reliable and portable numerical software.

Pioneering scientists including Brent, Cody, Kahan, and Kuki highlighted the relevant key concepts for designing an arithmetic that could be both useful for programmers and practical for implementers. These efforts resulted in the IEEE 754-1985 standard for radix-2 floating-point arithmetic, and its follower, the IEEE 854-1987 "radix-independent standard." The standardization process was expertly orchestrated by William Kahan. The IEEE 754-1985 standard was a key factor in improving the quality of the computational environment available to programmers. It has been revised during recent years, and its new version, the IEEE 754-2008 standard, was released in August 2008.

By carefully specifying the behavior of the arithmetic operators, the 754-1985 standard allowed researchers to design extremely smart yet portable algorithms; for example, to compute very accurate sums and dot products, and to formally prove some critical parts of programs. Unfortunately, the subtleties of the standard are hardly known by the nonexpert user. Even more worrying, they are sometimes overlooked by compiler designers. As a consequence, floating-point arithmetic is sometimes conceptually misunderstood and is often far from being exploited to its full potential.

This and the recent revision of the IEEE 754 standard led us to the decision to compile into a book selected parts of the vast knowledge on floating-point arithmetic. This book is designed for programmers of numerical applications, compiler designers, programmers of floating-point algorithms, designers of arithmetic operators, and more generally the students and researchers in numerical analysis who wish to more accurately understand a tool that they manipulate on an everyday basis. During the writing, we tried, whenever possible, to illustrate by an actual program the described techniques, in order to allow a more direct practical use for coding and design.

The first part of the book presents the history and basic concepts of floating-point arithmetic (formats, exceptions, correct rounding, etc.), and various aspects of the IEEE 754 and 854 standards and the new revised standard. The second part shows how the features of the standard can be used to develop smart and nontrivial algorithms. This includes summation algorithms, and division and square root relying on a fused multiply-add. This part also discusses issues related to compilers and languages. The third part then explains how to implement floating-point arithmetic, both in software (on an integer processor) and in hardware (VLSI or reconfigurable circuits). The fourth part is devoted to the implementation of elementary functions. The fifth part presents some extensions: certification of floating-point arithmetic and extension of the precision. The last part is devoted to perspectives and the Appendix.

## Acknowledgements

Some of our colleagues around the world and students from École Normale Supérieure de Lyon and Université de Lyon greatly helped us by reading preliminary versions of this book: Nicolas Bonifas, Pierre-Yves David, Jean-Yves l'Excellent, Warren Ferguson, John Harrison, Nicholas Higham, Nicolas Louvet, Peter Markstein, Adrien Panhaleux, Guillaume Revy, and Siegfried Rump. We thank them all for their suggestions and interest.

We have been very pleased working with our publisher, Birkhäuser Boston. Especially, we would like to thank Tom Grasso, Regina Gorenshteyn, and Torrey Adams for their help.

Jean-Michel Muller, Nicolas Brisebarre  
Florent de Dinechin, Claude-Pierre Jeannerod  
Vincent Lefèvre, Guillaume Melquiond  
Nathalie Revol, Damien Stehlé  
Serge Torres

Lyon, France  
July 2009

# List of Figures

2.1	Positive floating-point numbers for $\beta = 2$ and $p = 3$ . . . . .	18
2.2	Underflow before and after rounding. . . . .	19
2.3	The four rounding modes. . . . .	21
2.4	Relative error committed by rounding a real number to nearest floating-point number. . . . .	24
2.5	Values of ulp according to Harrison's definition. . . . .	33
2.6	Values of ulp according to Goldberg's definition. . . . .	33
2.7	Counterexample in radix 3 for a property of Harrison's ulp. . . . .	34
2.8	Conversion from ulps to relative errors. . . . .	38
2.9	Conversion from relative errors to ulps. . . . .	39
2.10	Converting from binary to decimal, and back. . . . .	42
2.11	Possible values of the binary ulp between two powers of 10. . . . .	43
2.12	Illustration of the conditions (2.10) in the case $b = 2^e$ . . . . .	47
3.1	Binary interchange floating-point formats. . . . .	81
3.2	Decimal interchange floating-point formats. . . . .	84
4.1	Independent operations in Dekker's product. . . . .	139
5.1	Convergence of iteration (5.4). . . . .	157
5.2	The various values that should be returned in round-to-nearest mode, assuming $q$ is within one $\text{ulp}(b/a)$ from $b/a$ . . . . .	164
5.3	Position of $Cx$ with respect to the result of Algorithm 5.2. . . . .	174
6.1	Boldo and Melquiond's algorithm for computing $\text{RN}(a+b+c)$ in radix-2 floating-point arithmetic. . . . .	200
8.1	Specification of the implementation of a FP operation. . . . .	240
8.2	Product-anchored FMA computation for normal inputs. . . . .	255
8.3	Addend-anchored FMA computation for normal inputs. . . . .	256
8.4	Cancellation in the FMA. . . . .	257
8.5	FMA $ab - c$ , where $a$ is the smallest subnormal, $ab$ is nevertheless in the normal range, $ c  <  ab $ , and we have an effective subtraction. . . . .	258

8.6	Significand alignment for the single-path algorithm. . . . .	260
9.1	Carry-ripple adder. . . . .	275
9.2	Decimal addition. . . . .	275
9.3	An implementation of the decimal DA box. . . . .	276
9.4	An implementation of the radix-16 DA box. . . . .	276
9.5	Binary carry-save addition. . . . .	277
9.6	Partial carry-save addition. . . . .	278
9.7	Carry-select adder. . . . .	279
9.8	Binary integer multiplication. . . . .	282
9.9	Partial product array for decimal multiplication. . . . .	283
9.10	A multipartite table architecture for the initial approximation of $1/x$ . . . . .	288
9.11	A dual-path floating-point adder. . . . .	289
9.12	Possible implementations of significand subtraction in the close path. . . . .	290
9.13	A dual-path floating-point adder with LZA. . . . .	292
9.14	Basic architecture of a floating-point multiplier without subnormal handling. . . . .	297
9.15	A floating-point multiplier using rounding by injection, without subnormal handling. . . . .	300
9.16	The classic single-path FMA architecture. . . . .	304
9.17	A pipelined SRT4 floating-point divider without subnormal handling. . . . .	306
9.18	Iterative accumulator. . . . .	313
9.19	Accumulator and post-normalization unit. . . . .	315
9.20	Accumulation of floating-point numbers into a large fixed-point accumulator. . . . .	315
9.21	The 2Sum and 2Mul operators. . . . .	319
11.1	The difference between $\ln$ and its degree-5 Taylor approximation in the interval $[1, 2]$ . . . . .	377
11.2	The difference between $\ln$ and its degree-5 minimax approximation in the interval $[1, 2]$ . . . . .	377
11.3	The $L^2$ approximation $p^*$ is obtained by projecting $f$ on the subspace generated by $T_0, T_1, \dots, T_n$ . . . . .	390
11.4	The $\exp(\cos(x))$ function and its degree-4 minimax approximation on $[0, 5]$ . . . . .	391
12.1	Example of an interval around $\hat{f}(x)$ containing $f(x)$ but no breakpoint. Hence, $\text{RN}(f(x)) = \text{RN}(\hat{f}(x))$ . . . . .	407
12.2	Example of an interval around $\hat{f}(x)$ containing $f(x)$ and a breakpoint. . . . .	408

12.3	Computing $P(1), P(2), P(3), \dots$ , for $P(X) = X^3$ with 3 additions per value. . . . .	433
12.4	The graph of $f$ (and $f^{-1}$ ) and a regular grid consisting of points whose coordinates are the breakpoints. . . . .	435
12.5	The integer grid and the segment $y = b - a.x$ ; the two-dimensional transformation modulo 1; and the representation of the left segment (corresponding to $x \in \mathbb{Z}$ ) modulo 1 as a circle. . . . .	438
12.6	Two-length configurations for $a = 17/45$ . . . . .	439
14.1	The representation of Algorithm 2Sum [180]. Here, $s = \text{RN}(a + b)$ , and $s + e = a + b$ exactly. . . . .	501
14.2	The representation of rounded-to-nearest floating-point addition and multiplication [180]. . . . .	501
14.3	SimpleAddQD: sum of two quadwords. . . . .	502
14.4	Graphic representation of Shewchuk's Scale-Expansion Algorithm [377] (Algorithm 14.10). . . . .	508
16.1	The lattice $\mathbb{Z}(2, 0) \oplus \mathbb{Z}(1, 2)$ . . . . .	524
16.2	Two bases of the lattice $\mathbb{Z}(2, 0) \oplus \mathbb{Z}(1, 2)$ . . . . .	525

# List of Tables

1.1	Results obtained by running Program 1.1 on a Pentium4-based workstation, using GCC and Linux. . . . .	10
2.1	Rounding a significand using the “round” and “sticky” bits.	22
2.2	ARRE and MRRE of various formats. . . . .	31
2.3	Converting from binary to decimal and back without error. .	44
3.1	Main parameters of the formats specified by the IEEE 754-1985 standard. . . . .	57
3.2	Sizes of the various fields in the formats specified by the IEEE 754-1985 standard, and values of the exponent bias. . . . .	57
3.3	Binary encoding of various floating-point data in single precision. . . . .	59
3.4	How to interpret the binary encoding of an IEEE 754-1985 floating-point number. . . . .	60
3.5	Extremal values in the IEEE 754-1985 standard. . . . .	61
3.6	The thresholds for conversion from and to a decimal string, as specified by the IEEE 754-1985 standard. . . . .	65
3.7	Correctly rounded decimal conversion range, as specified by the IEEE 754-1985 standard. . . . .	65
3.8	Comparison predicates and the four relations. . . . .	66
3.9	Floating-point from/to decimal string conversion ranges in the IEEE 854-1987 standard . . . . .	73
3.10	Correctly rounded conversion ranges in the IEEE 854-1987 standard. . . . .	73
3.11	Results returned by Program 3.1 on a 32-bit Intel platform. .	75
3.12	Results returned by Program 3.1 on a 64-bit Intel platform. .	76
3.13	Main parameters of the binary interchange formats of size up to 128 bits specified by the 754-2008 standard [187]. . . . .	81
3.14	Main parameters of the decimal interchange formats of size up to 128 bits specified by the 754-2008 standard [187]. . . . .	81
3.15	Width (in bits) of the various fields in the encodings of the binary interchange formats of size up to 128 bits [187]. . . . .	82

3.16	Width (in bits) of the various fields in the encodings of the decimal interchange formats of size up to 128 bits [187]. . . .	85
3.17	Decimal encoding of a decimal floating-point number (IEEE 754-2008). . . . .	87
3.18	Binary encoding of a decimal floating-point number (IEEE 754-2008). . . . .	88
3.19	Decoding the dectet $b_0b_1b_2 \cdots b_9$ of a densely packed decimal encoding to three decimal digits $d_0d_1d_2$ . . . . .	89
3.20	Encoding the three consecutive decimal digits $d_0d_1d_2$ , each of them being represented in binary by four bits, into a 10-bit dectet $b_0b_1b_2 \cdots b_9$ of a densely packed decimal encoding. . .	89
3.21	Parameters of the interchange formats. . . . .	93
3.22	Parameters of the binary256 and binary1024 interchange formats deduced from Table 3.21. . . . .	93
3.23	Parameters of the decimal256 and decimal512 interchange formats deduced from Table 3.21. . . . .	94
3.24	Extended format parameters in the IEEE 754-2008 standard. .	94
3.25	Minimum number of decimal digits in the decimal external character sequence that allows for an error-free write-read cycle, for the various basic binary formats of the standard. . .	100
3.26	Execution times of decimal operations on POWER6. . . . .	108
4.1	The four cases of Brent, Percival, and Zimmermann. . . . .	143
5.1	Quadratic convergence of iteration (5.4). . . . .	157
5.2	Comparison of various methods for checking Algorithm 5.2. .	176
6.1	Errors of various methods for $\sum x_i$ with $x_i = \text{RN}(\cos(i))$ . . .	198
6.2	Errors of various methods for $\sum x_i$ with $x_i = \text{RN}(1/i)$ . . . .	198
7.1	FLT_EVAL_METHOD macro values. . . . .	213
7.2	FORTTRAN allowable alternatives. . . . .	225
7.3	FORTTRAN forbidden alternatives. . . . .	226
8.1	Specification of addition/subtraction when both $x$ and $y$ are zero. . . . .	247
8.2	Specification of addition for positive floating-point data. . . .	247
8.3	Specification of subtraction for floating-point data of positive sign. . . . .	247
8.4	Specification of multiplication for floating-point data of positive sign. . . . .	251
8.5	Special values for $ x / y $ . . . . .	263
8.6	Special values for $\text{sqrt}(x)$ . . . . .	265
10.1	Standard integer encoding of binary32 data. . . . .	324

10.2	Some floating-point data encoded by $X$ . . . . .	330
11.1	Some worst cases for range reductions. . . . .	385
11.2	Degrees of minimax polynomial approximations for various functions and approximation ranges. . . . .	385
12.1	Actual and expected numbers of digit chains of length $k$ of the form $1000\cdots 0$ or $0111\cdots 1$ just after the $p$ -th bit of the infinitely precise significand of sines of floating-point numbers of precision $p = 16$ between $1/2$ and $1$ . . . . .	413
12.2	Actual and expected numbers of digit chains of length $k$ of the form $1000\cdots 0$ or $0111\cdots 1$ just after the $p$ -th bit of the infinitely precise significand of sines of floating-point numbers of precision $p = 24$ between $1/2$ and $1$ . . . . .	414
12.3	Length $k_{\max}$ of the largest digit chain of the form $1000\cdots 0$ or $0111\cdots 1$ just after the $p$ -th bit of the infinitely precise significands of sines and exponentials of floating-point numbers of precision $p$ between $1/2$ and $1$ , for various $p$ . . . . .	415
12.4	Some results for small values in the binary64 format, assuming rounding to nearest. . . . .	417
12.5	Some results for small values in the binary64 format, assuming rounding toward $-\infty$ . . . . .	418
12.6	Some bounds on the size of the largest digit chain of the form $1000\cdots 0$ or $0111\cdots 1$ just after the $p$ -th bit of the infinitely precise significand of $f(x)$ (or $f(x, y)$ ). . . . .	426
12.7	Worst cases for the function $1/\sqrt{x}$ , for binary floating-point systems and various values of the precision $p$ . . . . .	427
12.8	On the left, data corresponding to the current two-length configuration: the interval $I$ containing $b$ , its length, and the position of $b$ in $I$ . On the right, data one can deduce for the next two-length configuration: the new interval $I'$ containing $b$ and the position of $b$ in $I'$ . . . . .	440
12.9	Example with $a = 17/45$ and $b = 23.5/45$ . . . . .	442
12.10	Worst cases for functions $e^x$ , $e^x - 1$ , $2^x$ , and $10^x$ . . . . .	451
12.11	Worst cases for functions $\ln(x)$ and $\ln(1 + x)$ . . . . .	452
12.12	Worst cases for functions $\log_2(x)$ and $\log_{10}(x)$ . . . . .	453
12.13	Worst cases for functions $\sinh(x)$ and $\cosh(x)$ . . . . .	454
12.14	Worst cases for inverse hyperbolic functions. . . . .	455
12.15	Worst cases for the trigonometric functions. . . . .	456
12.16	Worst cases for the inverse trigonometric functions. . . . .	457
12.17	Longest runs $k$ of identical bits after the rounding bit in the worst cases of function $x^n$ , for $3 \leq n \leq 1035$ , in binary64. . . . .	459
14.1	Asymptotic complexities of multiplication algorithms. . . . .	514

## **Part I**

# **Introduction, Basic Definitions, and Standards**

# Chapter 1

## Introduction

REPRESENTING AND MANIPULATING real numbers efficiently is required in many fields of science, engineering, finance, and more. Since the early years of electronic computing, many different ways of approximating real numbers on computers have been introduced. One can cite (this list is far from being exhaustive): fixed-point arithmetic, logarithmic [220, 400] and semi-logarithmic [294] number systems, continued fractions [228, 424], rational numbers [227] and possibly infinite strings of rational numbers [275], level-index number systems [71, 318], fixed-slash and floating-slash number systems [273], and 2-adic numbers [425].

And yet, floating-point arithmetic is by far the most widely used way of representing real numbers in modern computers. Simulating an infinite, continuous set (the real numbers) with a finite set (the “machine numbers”) is not a straightforward task: clever compromises must be found between, e.g., speed, accuracy, dynamic range, ease of use and implementation, and memory cost. It appears that floating-point arithmetic, with adequately chosen parameters (radix, precision, extremal exponents, etc.), is a very good compromise for most numerical applications.

We will give a complete, formal definition of floating-point arithmetic in Chapter 3, but roughly speaking, a radix- $\beta$ , precision- $p$ , floating-point number is a number of the form

$$\pm m_0.m_1m_2 \cdots m_{p-1} \times \beta^e,$$

where  $e$ , called the *exponent*, is an integer, and  $m_0.m_1m_2 \cdots m_{p-1}$ , called the *significand*, is represented in radix  $\beta$ . The major purpose of this book is to explain how these numbers can be manipulated efficiently and safely.

### 1.1 Some History

Even if the implementation of floating-point arithmetic on electronic computers is somewhat recent, floating-point arithmetic itself is an old idea.

In *The Art of Computer Programming* [222], Donald Knuth presents a short history of floating-point arithmetic. He views the radix-60 number system of the Babylonians as some kind of early floating-point system. Since the Babylonians did not invent the zero, if the ratio of two numbers is a power of 60, then their representation in the Babylonian system is the same. In that sense, the number represented is the *significand* of a radix-60 floating-point representation of  $w$ .

A famous tablet from the Yale Babylonian Collection (YBC 7289) gives an approximation to  $\sqrt{2}$  with four sexagesimal places (the digits represented on the tablet are 1, 24, 51, 10). A photo of that tablet can be found in [434], and a very interesting analysis of the Babylonian mathematics related to YBC 7289 was done by Fowler and Robson [138].

The arithmetic of the slide rule, invented around 1630 by William Oughtred [433], can be viewed as another kind of floating-point arithmetic. Again, as with the Babylonian number system, we only manipulate significands of numbers (in that case, radix-10 significands).

The two modern co-inventors of floating-point arithmetic are probably Quevedo and Zuse. In 1914 Leonardo Torres y Quevedo described an electro-mechanical implementation of Babbage's Analytical Engine with floating-point arithmetic [341]. And yet, the first real, modern implementation of floating-point arithmetic was in Konrad Zuse's Z3 computer, built in 1941 [66]. It used a radix-2 floating-point number system, with 14-bit significands, 7-bit exponents and 1-bit sign. The Z3 computer had special representations for infinities and indeterminate results. These characteristics made the real number arithmetic of the Z3 much ahead of its time.

The Z3 was rebuilt recently [347]. Photographs of Konrad Zuse and the Z3 can be viewed at [http://www.computerhistory.org/projects/zuse\\_z23/](http://www.computerhistory.org/projects/zuse_z23/) and <http://www.konrad-zuse.de/>.

Readers interested in the history of computing devices should have a look at the excellent book by Aspray et al. [15].

Radix 10 is what humans use daily for representing numbers and performing paper and pencil calculations. Therefore, to avoid input and output radix conversions, the first idea that springs to mind for implementing automated calculations is to use the same radix.

And yet, since most of our computers are based on two-state logic, radix 2 (and, more generally, radices that are a power of 2) is by far the easiest to implement. Hence, choosing the right radix for the internal representation of floating-point numbers was not obvious. Indeed, several different solutions were explored in the early days of automated computing.

Various early machines used a radix-8 floating-point arithmetic: the PDP-10, and the Burroughs 570 and 6700 for example. The IBM 360 had a radix-16 floating-point arithmetic. Radix 10 has been extensively

used in financial calculations<sup>1</sup> and in pocket calculators, and efficient implementation of radix-10 floating-point arithmetic is still a very active domain of research [63, 85, 90, 91, 129, 414, 413, 428, 429]. The computer algebra system Maple also uses radix 10 for its internal representation of numbers. It therefore seems that the various radices of floating-point arithmetic systems that have been implemented so far have almost always been either 10 or a power of 2.

There has been a very odd exception. The Russian SETUN computer, built in Moscow University in 1958, represented numbers in radix 3, with digits  $-1, 0$ , and  $1$ . This “balanced ternary” system has several advantages. One of them is the fact that rounding to nearest is equivalent to truncation [222]. Another one [177] is the following. Assume you use a radix- $\beta$  fixed-point system, with  $p$ -digit numbers. A large value of  $\beta$  makes the implementation complex: the system must be able to “recognize” and manipulate  $\beta$  different symbols. A small value of  $\beta$  means that more digits are needed to represent a given number: if  $\beta$  is small,  $p$  has to be large. To find a compromise, we can try to minimize  $\beta \times p$ , while having the largest representable number  $\beta^p - 1$  (almost) constant. The optimal solution<sup>2</sup> will almost always be  $\beta = 3$ . See <http://www.computer-museum.ru/english/setun.htm> for more information on the SETUN computer.

Various studies (see references [44, 76, 232] and Chapter 2) have shown that radix 2 with the *implicit leading bit convention* (see Chapter 2) gives better worst-case or average accuracy than all other radices. This and the ease of implementation explain the current prevalence of radix 2.

The world of numerical computation changed much in 1985, when the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic was released [10]. This standard specifies various formats, the behavior of the basic operations and conversions, and exceptional conditions. As a matter of fact, the Intel 8087 mathematics co-processor, built a few years before in 1980, to be paired with the Intel 8088 and 8086 processors, was already extremely close to what would later become the IEEE 754-1985 standard. Now, most systems of commercial significance offer compatibility<sup>3</sup> with IEEE 754-1985. This has resulted in significant improvements in terms of accuracy, reliability, and portability of numerical software. William Kahan played a leading role in the conception of the IEEE 754-1985 standard and in the development of smart algorithms for floating-point arithmetic. His web page<sup>4</sup> contains much useful information.

---

<sup>1</sup>Financial calculations frequently require special rounding rules that are very tricky to implement if the underlying arithmetic is binary.

<sup>2</sup>If  $p$  and  $\beta$  were real numbers, the value of  $\beta$  that would minimize  $\beta \times p$  while letting  $\beta^p$  be constant would be  $e = 2.7182818 \dots$

<sup>3</sup>Even if sometimes you need to dive into the compiler documentation to find the right options: see Section 3.3.2 and Chapter 7.

<sup>4</sup><http://www.cs.berkeley.edu/~wkahan/>

IEEE 754-1985 only dealt with radix-2 arithmetic. Another standard, released in 1987, the IEEE 854-1987 Standard for Radix Independent Floating-Point Arithmetic [11], is devoted to both binary (radix-2) and decimal (radix-10) arithmetic.

IEEE 754-1985 and 854-1987 have been under revision since 2001. The new revised standard, called IEEE 754-2008 in this book, merges the two old standards and brings significant improvements. It was adopted in June 2008 [187].

## 1.2 Desirable Properties

Specifying a floating-point arithmetic (formats, behavior of operators, etc.) requires us to find compromises between requirements that are seldom fully compatible. Among the various properties that are desirable, one can cite:

- **Speed:** Tomorrow's weather must be computed in less than 24 hours;
- **Accuracy:** Even if speed is important, getting a wrong result right now is worse than getting the correct one too late;
- **Range:** We may need to represent large as well as tiny numbers;
- **Portability:** The programs we write on a given machine must run on different machines without requiring modifications;
- **Ease of implementation and use:** If a given arithmetic is too arcane, almost nobody will use it.

With regard to accuracy, the most accurate current physical measurements allow one to check some predictions of quantum mechanics or general relativity with a relative accuracy close to  $10^{-15}$ . This of course means that in some cases, we must be able to represent numerical data with a similar accuracy (which is easily done, using formats that are implemented on almost all current platforms). But this also means that we might sometimes be able to carry out computations that must end up with a relative error less than or equal to  $10^{-15}$ , which is much more difficult. Sometimes, one will need a significantly larger floating-point format or smart "tricks" such as those presented in Chapter 4.

An example of a huge calculation that requires much care was carried out by Laskar's team at the Paris observatory [243]. They computed long-term numerical solutions for the insolation quantities of the Earth (very long-term, ranging from  $-250$  to  $+250$  millions of years from now).

In other domains, such as number theory, some multiple-precision computations are indeed carried out using a very large precision. For instance,

in 2002, Kanada's group computed 1241 billion decimal digits of  $\pi$  [19], using the two formulas

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

These last examples are extremes. One should never forget that with 50 bits, one can express the distance from the Earth to the Moon with an error less than the thickness of a bacterium. It is very uncommon to need such an accuracy on a final result and, actually, very few physical quantities are defined that accurately.

## 1.3 Some Strange Behaviors

Designing efficient and reliable hardware or software floating-point systems is a difficult and somewhat risky task. Some famous bugs have been widely discussed; we recall some of them below. Also, even when the arithmetic is not flawed, some strange behaviors can sometimes occur, just because they correspond to a numerical problem that is intrinsically difficult. All this is not surprising: mapping the continuous real numbers on a finite structure (the floating-point numbers) cannot be done without any trouble.

### 1.3.1 Some famous bugs

- The divider of the first version of the Intel Pentium processor, released in 1994, was flawed [290, 122]. In extremely rare cases, one would get three correct decimal digits only. For instance, the computation of

$$8391667/12582905$$

would give  $0.666869\dots$  instead of  $0.666910\dots$ .

- With release 7.0 of the computer algebra system Maple, when computing

$$\frac{1001!}{1000!}$$

we would get 1 instead of 1001.

- With the previous release (6.0) of the same system, when entering

$$21474836480413647819643794$$

you would get

$$413647819643790) + ' - - .(- - .($$

- Kahan [208] mentions some strange behavior of some versions of the Excel spreadsheet. They seem to be due to an attempt to mimic a decimal arithmetic with an underlying binary one.

An even more striking behavior happens with some early versions of Excel 2007: When you try to compute

$$65536 - 2^{-37}$$

the displayed result is 100001. This is an error in the binary-to-decimal conversion used for displaying that result: the internal binary value is correct, if you add 1 to that result you get 65537. An explanation can be found at <http://blogs.msdn.com/excel/archive/2007/09/25/calculation-issue-update.aspx>, and a patch is available from <http://blogs.msdn.com/excel/archive/2007/10/09/calculation-issue-update-fix-available.aspx>

- Some bugs do not require any programming error: they are due to poor specifications. For instance, the Mars Climate Orbiter probe crashed on Mars in September 1999 because of an astonishing mistake: one of the teams that designed the numerical software assumed the unit of distance was the meter, while another team assumed it was the foot [7, 306].

Very similarly, in June 1985, a space shuttle positioned itself to receive a laser beamed from the top of a mountain that was supposedly 10,000 miles high, instead of the correct 10,000 feet [7].

Also, in January 2004, a bridge between Germany and Switzerland did not fit at the border because the two countries use a different definition of the sea level.<sup>5</sup>

### 1.3.2 Difficult problems

Sometimes, even with a correctly implemented floating-point arithmetic, the result of a computation is far from what could be expected.

#### A sequence that seems to converge to a wrong limit

Consider the following example, due to one of us [289] and analyzed by Kahan [208, 291]. Let  $(u_n)$  be the sequence defined as

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}. \end{cases} \quad (1.1)$$

<sup>5</sup>See <http://www.spiegel.de/panorama/0,1518,281837,00.html>.

One can easily show that the limit of this sequence is 6. And yet, on any system with any precision, the sequence will seem to go to 100.

For example, Table 1.1 gives the results obtained by compiling Program 1.1 and running it on a Pentium4-based workstation, using the GNU Compiler Collection (GCC) and the Linux system.

```

#include <stdio.h>

int main(void)
{
    double u, v, w;
    int i, max;

    printf("n =");
    scanf("%d",&max);
    printf("u0 = ");
    scanf("%lf",&u);
    printf("u1 = ");
    scanf("%lf",&v);
    printf("Computation from 3 to n:\n");
    for (i = 3; i <= max; i++)
    {
        w = 111. - 1130./v + 3000./(v*u);
        u = v;
        v = w;
        printf("u%d = %1.17g\n", i, v);
    }
    return 0;
}

```

*Program 1.1: A C program that is supposed to compute sequence  $u_n$  using double-precision arithmetic. The obtained results are given in Table 1.1.*

The explanation of this weird phenomenon is quite simple. The general solution for the recurrence

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}$$

is

$$u_n = \frac{\alpha \cdot 100^{n+1} + \beta \cdot 6^{n+1} + \gamma \cdot 5^{n+1}}{\alpha \cdot 100^n + \beta \cdot 6^n + \gamma \cdot 5^n},$$

where  $\alpha$ ,  $\beta$ , and  $\gamma$  depend on the initial values  $u_0$  and  $u_1$ . Therefore, if  $\alpha \neq 0$  then the limit of the sequence is 100, otherwise (assuming  $\beta \neq 0$ ), it is 6. In the present example, the starting values  $u_0 = 2$  and  $u_1 = -4$  were chosen so that  $\alpha = 0$ ,  $\beta = -3$ , and  $\gamma = 4$ . Therefore, the “exact” limit of  $u_n$  is 6. And yet, when computing the values  $u_n$  in floating-point arithmetic using (1.1), due to the various rounding errors, even the very first computed terms become slightly different from the exact terms. Hence, the value  $\alpha$  corresponding to

$n$	Computed value	Exact value
3	18.5	18.5
4	9.378378378378379	9.3783783783783784
5	7.8011527377521679	7.8011527377521613833
6	7.1544144809753334	7.1544144809752493535
11	6.2744386627644761	6.2744385982163279138
12	6.2186967691620172	6.2186957398023977883
16	6.1661267427176769	6.0947394393336811283
17	7.2356654170119432	6.0777223048472427363
18	22.069559154531031	6.0639403224998087553
19	78.58489258126825	6.0527217610161521934
20	98.350416551346285	6.0435521101892688678
21	99.898626342184102	6.0360318810818567800
22	99.993874441253126	6.0298473250239018567
23	99.999630595494608	6.0247496523668478987
30	99.99999999998948	6.0067860930312057585
31	99.9999999999943	6.0056486887714202679

Table 1.1: Results obtained by running Program 1.1 on a Pentium4-based workstation, using GCC and the Linux system, compared to the exact values of sequence  $u_n$ .

these computed terms is very tiny, but nonzero. This suffices to make the computed sequence “converge” to 100.

### The Chaotic Bank Society

Recently, Mr. Gullible went to the Chaotic Bank Society, to learn more about the new kind of account they offer to their best customers. He was told:

You first deposit  $\$e - 1$  on your account, where  $e = 2.7182818 \dots$  is the base of the natural logarithms. The first year, we take \$1 from your account as banking charges. The second year is better for you: We multiply your capital by 2, and we take \$1 of banking charges. The third year is even better: We multiply your capital by 3, and we take \$1 of banking charges. And so on: The  $n$ -th year, your capital is multiplied by  $n$  and we just take \$1 of charges. Interesting, isn't it?

Mr. Gullible wanted to secure his retirement. So before accepting the offer, he decided to perform some simulations on his own computer to see what his capital would be after 25 years. Once back home, he wrote a C program (Program 1.2).

```

#include <stdio.h>

int main(void)
{
    double account = 1.71828182845904523536028747135;
    int i;
    for (i = 1; i <= 25; i++)
    {
        account = i*account - 1;
    }
    printf("You will have $%1.17e on your account.\n", account);
}

```

Program 1.2: Mr. Gullible's C program.

On his computer (with an Intel Xeon processor, and GCC on Linux, but strange things would happen with any other equipment), he got the following result:

You will have \$1.20180724741044855e+09 on your account.

So he immediately decided to accept the offer. He will certainly be sadly disappointed, 25 years later, when he realizes that he actually has around \$0.0399 on his account.

What happens in this example is easy to understand. If you call  $a_0$  the amount of the initial deposit and  $a_n$  the capital after the end of the  $n$ -th year, then

$$\begin{aligned}
 a_n &= n! \times \left( a_0 - 1 - \frac{1}{2!} - \frac{1}{3!} - \cdots - \frac{1}{n!} \right) \\
 &= n! \times \left( a_0 - (e - 1) + \frac{1}{(n+1)!} + \frac{1}{(n+2)!} + \frac{1}{(n+3)!} + \cdots \right),
 \end{aligned}$$

so that:

- if  $a_0 < e - 1$ , then  $a_n$  goes to  $-\infty$ ;
- if  $a_0 = e - 1$ , then  $a_n$  goes to 0;
- if  $a_0 > e - 1$ , then  $a_n$  goes to  $+\infty$ .

In our example,  $a_0 = e - 1$ , so the *exact* sequence  $a_n$  goes to zero. This explains why the exact value of  $a_{25}$  is so small. And yet, even if the arithmetic operations were errorless (which of course is not the case), since  $e - 1$  is not exactly representable in floating-point arithmetic, the *computed* sequence will go to  $+\infty$  or  $-\infty$ , depending on rounding directions.