RETROSPECTIVE:

How to Print Floating-Point Numbers Accurately

Guy L. Steele Jr.
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803 USA
guy.steele@sun.com

Jon L White
Lisp Wizard
2001 The Forward March of Technology Drive
Oceanview, Kansas 99999 USA
jonl@jonl.us

1. History

Our PLDI paper [28] was almost 20 years in the making.

How should the result of dividing `1.0` by `10.0` be printed? In 1970, one usually got "`0.0999999`" or "`0.099999994`"; why not "`0.1`"? (The problem of binary-to-decimal conversion is amazingly tricky, even for fixed-point fractions, let alone floating-point numbers [31][22][20][12].) Some programming systems of the day required a prespecified limit for the number of decimal digits to be printed for a binary floating-point number. If, in our example, the limit were set to nine decimal places, then the printed result might well be `0.099999994`; but if the limit were set to seven decimal places, then poorly implemented printers would produce `0.0999999` and those that took the trouble to perform decimal rounding would produce either `0.1000000` or `0.1`.

What is the correct number of digits to produce if the user doesn't specify? If a system prints too many digits, the excess digits may be "garbage," reflecting more information than the number actually contains; if a system prints too few digits, the result will be wrong in a stronger sense: converting the decimal representation back to binary may not recover the original binary value.

It really bugged one of us (White) that systems of the day not only produced incorrect decimal representations of binary numbers, but provided no guaranteed bound on how wrong they might be. In early 1971, he began to analyze the machine-language algorithm used in the implementation of MacLisp [23][27] to convert PDP-10 floating-point numbers into decimal notation. He also investigated the behavior of other language systems, both those that printed a variable number of digits and those that printed a fixed number of digits but failed to perform decimal rounding correctly.

Using the "bignum" (high-precision integer) facilities that had recently been added to MacLisp for other purposes, White investigated the exact representations of certain floating point numbers that seemed to print out with too many decimal digits. He discovered that virtually all of the systems he examined suffered from one or more of the following problems on many floating-point values: (a) printing many more decimal digits than were necessary, (b) printing decimal values that differed from the binary values by many units in the last place (ulps), and (c) exhibiting substantial "drift" when a value is printed, then read back in, the new value printed and again read back in, and so on.

White found that the programmers or authors of these systems invariably claimed that their systems must be producing "correct" results because (1) they produced lots of decimal digits, making no claim of producing "shortest" representations, and/or (2) they had coded some variant of Taranto's algorithm [31]—"I found it in Knuth!" [18, ex. 4.4–3]. Some also dismissed examples of discrepancies with hand-waving generalizations like those found in Knuth ("… floating point arithmetic is inherently approximate" [18, section 4.2.1]) and later in Steele ("In general, computations with floating-point numbers are only approximate." [29, section 12.1]. But White saw no reason why the printing problem could not be solved exactly, and he set out to find a way.

One suggested plan for obtaining the shortest output that correctly preserves the floating-point value was simply to produce many more decimal digits than necessary; then one would "round backwards" in decimal, converting the decimal string back into binary at each step, until the shortest string of digits was found that would convert back into the original number. However, this approach is quite inefficient. It fails to take advantage of information that is incrementally available during the steps of digit generation and that adequately characterizes the difference between what has already (incrementally) been outputted and the original number.

The first attempt to take advantage of such information was by White, later in 1971. By 1972, when Steele joined White at MIT to work on MacLisp, there was a rudimentary version of the algorithm in the MacLisp `PRINT` function, better than previous printing algorithms but not yet correct in all cases. White then decided to try tracking the error propagation by watching what happened to a value equal to $\frac{1}{2}$ ulp, subjected to the same arithmetic operations that were being performed on the fraction being printed. This line of investigation led to the use of the variable M in our algorithms. (We called this value "M" because White thought of it as defining a "mask," the complement of the interval $[M, 1-M]$, that filters ("masks") out fractional values for which enough digits had been generated and passes through only fractions for which more digits are required, namely those lying in the gap $[M, 1-M]$. As M grows, the mask grows and the interval at its center shrinks, until eventually M exceeds $\frac{1}{2}$, the gap shrinks away to nothing, and no remaining fraction can pass through the mask.)

When Steele left MIT in 1980 to go to Carnegie-Mellon University, MacLisp was using the algorithm Dragon2, with double-precision floating-point prescaling of values that were large or small enough to require "`E`" notation. By about this time Steele had written a very early draft of the PLDI paper. (He wrote it in TEX, so it must have been after Fall 1978, when he first ported TEX from Stanford to MIT, and it must have been prior to the 1981 publication of the second edition of "Knuth Volume 2" [19], which mentions this unpublished draft in the answer to exercise 4.4–3.)

Steele continued to polish the algorithm, off and on, during the early 1980s. His principal contributions were (1) to replace floating-point prescaling with implicit rational prescaling, thus introducing the scale factor S to the algorithm (and incidentally com-

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

mitting to the use of bignum arithmetic in the printing algorithm itself); (2) to use two mask quantities M^+ and M^- rather than just one; (3) to separate the digit-generation process from the formatting process, organizing them as coroutines; (4) to provide a proof of correctness for the digit-generation process; and (5) to code the whole thing up (in Pascal), including many different formatters. (The PL/I-style "picture" formatter was omitted from the PLDI paper for lack of space, alas.) Steele finished all this by 1985.

During the 1980s, White investigated the question of whether one could use limited-precision arithmetic after all rather than bignums. He had earlier proved by exhaustive testing that just 7 extra bits suffice for correctly printing 36-bit PDP-10 floating-point numbers, if powers of ten used for prescaling are precomputed using bignums and rounded just once. But can one derive, without exhaustive testing, the necessary amount of extra precision solely as a function of the precision and exponent range of a floating-point format? This problem is still open, and appears to be very hard.

It weighed on Steele's mind that Knuth had shown confidence in him by citing—in a book!—a paper that didn't really exist yet. Eventually Steele summoned the effort to complete the paper and submit it to the 1990 PLDI, just to rid himself of the nagging burden. Imagine our surprise when we discovered that Will Clinger had written a paper on the decimal-to-binary problem and submitted it to the same conference! (By the way, the third edition of "Knuth Volume 2" [21] cites our PLDI paper, so all is well now.)

Since our paper was published, there seem to have been exactly two follow-up papers [9][2] of any consequence. Both provided important practical improvements on our work; we recommend them to interested readers and especially to implementors.

2. Influence on Programming Languages

Since our paper was published, it has had a clear influence on the specification and implementation of many programming languages.

We had coded an early version of this algorithm as part of the runtime system for MacLisp while we were at MIT in the late 1970s; this version was not completely accurate because it first used floating-point operations to scale the number to be printed by a power of 10 if the number was very large or small, but it was completely accurate for numbers not requiring such prescaling.

The MacLisp successor Lisp Machine Lisp, also developed at MIT, was influenced by our work [32, p. 281]:

> The number of digits printed is the "correct" number; no information present in the flonum is lost, and no extra trailing digits are printed that do not represent information in the flonum. Feeding the [printed representation] of a flonum back to the reader is always supposed to produce an equal flonum.

The documentation for the commercial version of this language, Symbolics' Zetalisp, has similar wording [30, p. 15].

After publication of our paper, the idea began to spread, to the language Id, another MIT project, in 1991 [1, p. 15]:

> Utilities for accurate reading and printing of double precision floating point numbers have been implemented. The floating point reader [3] has been implemented in Common Lisp and Id 90, the floating point printer [28] in Common Lisp.

and to Modula-3 [14, p. 15]:

> The idea of converting to decimal by retaining just as many digits as are necessary to convert back to binary exactly was popularized by Guy L. Steele Jr. and Jon L White [28]. David M. Gay pointed out the importance … of demanding that the conversion to binary handle mid-point cases by a known rule [9].

and to the fourth revision of the report on Scheme, which cites [28] and [3] and also remarks [24, p. 24]:

> … the result [of a binary-to-decimal conversion] is expressed using the minimum number of digits …

The fifth revision of the report [17] has similar wording but cites [2] rather than [28].

Steele had a hand in establishing conversion accuracy requirements in *The Java Language Specification* [11, pp. 506–507], which requires that all binary-to-decimal and decimal-to-binary conversions be correctly rounded and that binary-to-decimal conversions generate as few digits as possible.

The description of the "basis library" for Standard ML says [25]:

> `toDecimal` should produce only as many digits as are necessary for `fromDecimal` to convert back to the same number, i.e., for any `Normal` or `SubNormal` real value `r`, we have: `fromDecimal (toDecimal r) = r` … Algorithms for accurately and efficiently converting between binary and decimal real representations are readily available, e.g., see the technical report by [9].

Soon after that, the language Limbo adopted accurate base conversion as one of its improvements over C [13, p. 271]:

> The most important numerical development at the language level recently has been accurate binary/decimal conversion [3][9][28]. Thus printing a real using `%g` and reading it on a different machine guarantees recovering identical bits.

As for Haskell, a comment that appears in the code for the function `floatToDigits` in the Haskell 98 library [16, p. 14] says that the code is based on Burger and Dybvig's work [2].

Borneo is a numerical programming language [4, pp. 7, 9]:

> Although published algorithms exist for both correctly rounded input [3] and output [28], conversion problems persist. Correctly rounded algorithms are also acceptably fast for common cases [2], [9]. While working on the BEEF tests for transcendental functions, it was discovered that the Turbo C 1.0 compiler did not convert 11.0 exactly into a floating point number equal to 11! … Java requires correctly rounded decimal to binary and binary to decimal conversion … Borneo maintains Java's base conversion requirements …

Steele served as Project Editor for the first edition of the ECMA standard for the web browser scripting language ECMAScript (more popularly known as JavaScript), which recommends, but does not require, accurate conversions [5, p. 36]:

> NOTE: The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard: If x is any number value other than 0, then ToNumber(ToString(x)) is exactly the same number value as x.… Implementors of ECMAScript may find useful the paper and code written by David M. Gay for binary-to-decimal conversion of floating-point numbers [9].

A recent extension of the GNU Fortran compiler makes use of accurate conversion techniques [26, p. 319]:

> … support routines for performing conversion between character strings and intervals … were developed based on routines for floating-point input [9] and floating-point output [2].

Finally, David M. Gay himself has been working on an algebraic modeling language called AMPL [10, p. 3]:

> AMPL and its solver interface library use correctly rounded binary↔decimal conversions, which is now possible on all machines where AMPL has run other than old Cray machines. Details are described in Gay [9]. Part of the reason for mentioning binary↔decimal conversions here is to point out a recent extension to Gay [9] that carries out correctly rounded

conversions for other arithmetics with properties similar to binary IEEE arithmetic. This includes correct directed roundings and rounding of a decimal string to a floating-point interval of width at most one unit in the last place, both of which are obviously useful for rigorous interval computations. There is no paper yet about this work, but the source files are available as ftp://netlib.bell-labs.com/netlib/fp/gdtoa.tgz which includes a README file for documentation.

The IEEE 754 standard for floating-point arithmetic [15] has a conversion accuracy requirement, but a liberal one. Now that the practicality of our approach has been established, we have high hopes that an upcoming revision of that standard will adopt a requirement of full conversion accuracy.

3. Why "Dragon"?

A very few readers have wondered why the principal algorithms in our work are named "Dragon2" and "Dragon3" and "Dragon4"; this was an obscure joke (entirely Steele's fault) alluding to the "dragon curves" discovered and analyzed by John E. Heighway, Bruce A. Banks, and William G. Harter and reported on by Martin Gardner [7][8]. The initial letters in the multiword description of a "Dragon" algorithm form a sequence of letters 'F' and 'P' that represents the sequence of (valley) Folds and (mountain) Peaks in a piece of paper that has been folded to form a dragon curve.

REFERENCES

[1] Boughton, G. A., editor. *Computation Structures Group Progress Report 1990–91*. CSG Memo 337. MIT Laboratory for Computer Science (Cambridge, MA, June 1991).

[2] Burger, Robert G., and Dybvig, R. Kent. Printing floating-point numbers quickly and accurately. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design and Implementation*. ACM (Philadelphia, PA, June 1996), 108–116.

[3] Clinger, William D. How to read floating point numbers accurately. In *Proc. ACM SIGPLAN '90 Conf. Prog. Lang. Design and Implementation*. ACM (White Plains, NY, June 1990), 92–101. *ACM SIGPLAN Notices* **25**, 6 (June 1990).

[4] Darcy, Joseph D. *Borneo 1.0.2: Adding IEEE 754 floating point support to Java*. U. California, Berkeley, May 1998.

[5] *ECMAScript Language Specification*, third edition. ECMA (December 1999). Standard ECMA-262.

[6] Gansner, Emden R., and Reppy, John H. *The Standard ML Basis Manual*. Cambridge University Press (New York, 2003). Not yet published—available from October 2003.

[7] Gardner, Martin. Mathematical games. *Scientific American 216*, 3 (March 1967), 124–125; *216*, 4 (April 1967), 118–120; *217*, 1 (July 1967), 115.

[8] Gardner, Martin. *Mathematical Magic Show.* Vintage (New York, 1978), 207–209, 215–220.

[9] Gay, David M. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*. Numerical Analysis Manuscript 90-10. AT&T Bell Laboratories (Murray Hill, NJ, November 1990).

[10] Gay, David M. *Symbolic-Algebraic Computations in a Modeling Language for Mathematical Programming.* Technical Report 00-3-02. Computing Sciences Research Center, Bell Laboratories (Murray Hill, NJ, July 2000).

[11] Gosling, James, Joy, Bill, and Steele, Guy. *The Java Language Specification*. Addison-Wesley (Reading, MA, 1996).

[12] Gries, David. Binary to decimal, one more time. In Feijen, W. H. J., van Gasteren, A. J. M., Gries, D., and Misra, J., eds., *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag (Berlin, 1990), 141–148.

[13] Grosse, Eric. Real Inferno. In Boisvert, Ronald F., editor, *The Quality of Numerical Software: Assessment and Enhancement: Proc. IFIP TC2/WG 2.5 Working Conf., Oxford, United Kingdom, 8–12 July 1996*. Chapman Hall on behalf of IFIP (London, 1997), 270–279.

[14] Horning, Jim, Kalsow, Bill, McJones, Paul, and Nelson, Greg. *Some Useful Modula-3 Interfaces*. Memo 113. Digital Equipment Corporation Systems Research Center (Palo Alto, CA, December 1993).

[15] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985 edition. IEEE (New York, 1985).

[16] Standard libraries for the Haskell 98 programming language. http://www.haskell.org/definition/haskell98-library.pdf, February 1999.

[17] Kelsey, Richard, Rees, Jonathan, Clinger, William, et al. The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (September 1998), 26–76.

[18] Knuth, Donald E. *Seminumerical Algorithms*. Addison-Wesley (Reading, MA, 1969).

[19] Knuth, Donald E. *Seminumerical Algorithms (Second Edition)*. Addison-Wesley (Reading, MA, 1981).

[20] Knuth, Donald E. A simple program whose proof isn't. In Feijen, W. H. J., van Gasteren, A. J. M., Gries, D., and Misra, J., eds., *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag (Berlin, 1990), 233–242.

[21] Knuth, Donald E. *Seminumerical Algorithms (Third Edition)*. Addison-Wesley (Reading, MA, 1998).

[22] Matula, David W. In-and-out conversions. *Communications of the ACM 11*, 1 (January 1968), 47–50.

[23] Moon, David A. *MacLISP Reference Manual*. MIT Project MAC (Cambridge, MA, April 1974).

[24] Rees, Jonathan, Clinger, William, et al. The revised^4 report on the algorithmic language Scheme. *ACM SIGPLAN Lisp Pointers 4*, 3 (July–September 1991), 1–55.

[25] Reppy, John H., et al. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/pages/real.html, October 1997. To be published as [6].

[26] Schulte, M. J., Zelov, V. A., Akkas, A., and Burley, J. C. The interval-enhanced GNU Fortran compiler. In Csendes, Tibor, ed., *Developments in Reliable Computing*. Kluwer (Dordrecht, Netherlands, 1999), 311–322.

[27] Steele, Jr., Guy L., and Gabriel, Richard P. The evolution of Lisp, pages 233–330. In *History of Programming Languages*. ACM Press (New York, 1996), 233–330.

[28] Steele, Jr., Guy L., and White, Jon L. How to print floating-point numbers accurately. In *Proc. ACM SIGPLAN '90 Conf. Prog. Lang. Design and Implementation*. ACM (White Plains, NY, June 1990), 112–126. *ACM SIGPLAN Notices* **25**, 6 (June 1990).

[29] Steele, Guy L., Jr., Fahlman, Scott E., Gabriel, Richard P., Moon, David A., and Weinreb, Daniel L. *Common Lisp: The Language*. Digital Press (Burlington, MA, 1984).

[30] *Reference Guide to Symbolics-Lisp*. Symbolics, Inc. (Cambridge, MA, March 1985).

[31] Taranto, Donald. Binary conversion, with fixed decimal precision, of a decimal fraction. *Communications of the ACM 2*, 7 (July 1959), 27.

[32] Weinreb, Daniel, and Moon, David. *LISP Machine Manual, Third Edition*. MIT Artificial Intelligence Laboratory (Cambridge, MA, March 1981).

How to Print Floating-Point Numbers Accurately

Guy L. Steele Jr.
Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142
gls@think.com

Jon L White
Lucid, Inc.
707 Laurel Street
Menlo Park, California 94025
jonl@lucid.com

Abstract

We present algorithms for accurately converting floating-point numbers to decimal representation. The key idea is to carry along with the computation an explicit representation of the required rounding accuracy.

We begin with the simpler problem of converting fixed-point fractions. A modification of the well-known algorithm for radix-conversion of fixed-point fractions by multiplication explicitly determines when to terminate the conversion process; a variable number of digits are produced. The algorithm has these properties:

- No information is lost; the original fraction can be recovered from the output by rounding.
- No "garbage digits" are produced.
- The output is correctly rounded.
- It is never necessary to propagate carries on rounding.

We then derive two algorithms for *free-format* output of floating-point numbers. The first simply scales the given floating-point number to an appropriate fractional range and then applies the algorithm for fractions. This is quite fast and simple to code but has inaccuracies stemming from round-off errors and oversimplification. The second algorithm guarantees mathematical accuracy by using multiple-precision integer arithmetic and handling special cases. Both algorithms produce no more digits than necessary (intuitively, the "1.3 prints as 1.2999999" problem does not occur).

Finally, we modify the free-format conversion algorithm for use in *fixed-format* applications. Information may be lost if the fixed format provides too few digit positions, but the output is always correctly rounded. On the other hand, no "garbage digits" are ever produced, even if the fixed format specifies too many digit positions (intuitively, the "4/3 prints as 1.3333333328366279602" problem does not occur).

1 Introduction

Isn't it a pain when you ask a computer to divide 1.0 by 10.0 and it prints 0.0999999? Most often the arithmetic is not to blame, but the printing algorithm.

Most people use decimal numerals. Most contemporary computers, however, use a non-decimal internal representation, usually based on powers of two. This raises the problem of conversion between the internal representation and the familiar decimal form.

The external decimal representation is typically used in two different ways. One is for communication with people. This divides into two cases. For *fixed-format output*, a fixed number of digits is chosen before the conversion process takes place. In this situation the primary goal is controlling the precise number of characters produced so that tabular formats will be correctly aligned. Contrariwise, in *free-format output* the goal is to print no more digits than are necessary to communicate the value. Examples of such applications are Fortran list-directed output and such interactive language systems as Lisp and APL. Here the number of digits to be produced may be dependent on the value to be printed, and so the conversion process itself must determine how many digits to output.

The second way in which the external decimal representation is used is for inter-machine communication. The decimal representation is a convenient and conventional machine-independent format for transferring numerical information from one computer to another. Using decimal representation avoids incompatibilities of word length, field format, radix (binary versus hexadecimal, for example), sign representation, and so on. Here the main objective is to transport the numerical value as accurately as possible; that the representation is also easy for people to read is a bonus. (The IEEE 754 floating-point standard [IEEE85] has ameliorated this problem by providing standard binary formats, but the VAX, IBM 370, and Cray-1 are still with us and will be for yet a while longer.)

We deal here with the *output problem*: methods of converting from the internal representation to the external representation. The corresponding *input problem* is also of interest and is addressed by another paper in this conference [Clinger90]. The two problems are not quite identical because we make the asymmetrical assumption that the internal representation has a fixed number of digits but the external representation may have a varying number of digits. (Compare this with Matula's excellent work, in which both representations are assumed to be of fixed precision [Matula68] [Matula70].)

In the following discussion we assume the existence of a program that solves the input problem: it reads a number in external representation and produces the input representation whose exact value is closest, of all possible internal representation values, to the exact numerical value of the external number. In other words, this ideal input routine is assumed to round correctly in all cases.

We present algorithms for fixed-point-fraction output and floating-point output that decide dynamically what is an appropriate number of output digits to produce for the external representation. We recommend the last algorithm, Dragon4, for general use in compilers and run-time systems; it is completely accurate and accommodates a wide variety of output formats.

We express the algorithms in a general form for converting from any radix b (the *internal* radix) to any other radix B (the *external* radix); b and B may be any two integers greater than 1, and either may be larger than the other. The reader may find it helpful to think of B as being 10, and b as being either 2 or 16. Informally we use the terms "decimal" and "external radix" interchangeably, and similarly "binary" and "internal radix". We also speak interchangeably of digits being "output" or "printed".

2 Properties of Radix Conversion

We assume that a number to be converted from one radix to another is perfectly accurate and that the goal is to express that exact value. Thus we ignore issues of errors (such as floating-point round-off or truncation) in the calculation of that value.

The decimal representation used to communicate a numerical value should be accurate. This is especially important when transferring values between computers. In particular, if the source and destination computers use identical floating-point representations, we would like the value to be communicated exactly; the destination computer should reconstruct the exact bit pattern that the source computer had. In this case we require that conversion from internal form to external form and back be an identity function; we call this the *internal identity requirement.* We would also like conversion from external form to internal form and back to be an identity; we call this the *external identity requirement.* However, there are some difficulties with these requirements.

The first difficulty is that an exact representation is not always possible. A finite *integer* in any radix can be converted into a finite integer in any other radix. This is not true, however, of finite-length *fractions*; a fraction that has a finite representation in one radix may have an indefinitely repeating representation in another radix. This occurs only if there is some prime number p that divides the one radix but not the other; then $1/p$ (among other numbers) has a finite representation in the first radix but not in the second. (It should be noted, by the way, that this relationship between radices is different from the relationship of *incommensurability* as Matula defines it [Matula70]. For example, radices 10 and 20 are incommensurable, but there is no prime that divides one and not the other.)

It does happen that any binary, octal, or hexadecimal number has a finite decimal representation (because there is no prime that divides 2^k but not 10). Therefore when $B = 10$ and b is a power of two we can guarantee the internal identity requirement by printing an exact decimal representation of a given binary number and using an ideal rounding input routine.

This solution is gross overkill, however. The exact decimal representation of an n-bit binary number may require n decimal digits. However, as a rule fewer than n digits are needed by the rounding input routine to reconstruct the exact binary value. For example, to print a binary floating-point number with a 27-bit fraction requires up to 30 decimal digits. (A 27-bit fraction can represent a 30-bit binary fraction whose decimal representation has a non-zero leading digit; this is discussed further below.) However, 10 decimal digits always suffice to communicate the value accurately enough for the input routine to reconstruct the 27-bit binary representation [Matula68], and many particular values can be expressed accurately with fewer than 10 digits. For example, the 27-bit binary floating-point number $.110011001100110011001100110_2 \times 2^{-3}$, which is equal to the 29-bit binary fixed-point fraction $.001100110011001100110011001_2$, is also equal to the decimal fraction .09999999962747097015380859375, which has 29 digits. However, the decimal fraction 0.1 is closer in value to the binary number than to any other 27-bit floating-point binary number, and so suffices to satisfy the internal identity requirement. Moreover, 0.1 is more concise and more readable.

We would like to produce enough digits to preserve the information content of a binary number, but no more. The difficulty is that the number of digits needed

depends not only on the precision of the binary number but on its value.

Consider for a moment the superficially easier problem of converting a binary floating-point number to octal. Suppose for concreteness that our binary numbers have a six-bit significand. It would seem that two octal digits should be enough to express the value of the binary number, because one octal digit expresses three bits of information. However, the precise octal representation of the binary floating-point number $.101101_2 \times 2^{-1}$ is $.264_8 \times 8^0$. The exponent of the binary floating-point number specifies a shifting of the significand so that the binary point is "in the middle" of an octal digit. The first octal digit thus represents only two bits of the original binary number, and the third only one bit. We find that three octal digits are needed because of the difference in "graininess" of the exponent (shifting) specifications of the two radices.

The number N of radix-B digits that may be required in general to represent an n-digit radix-b floating-point number is derived in [Matula68]. The condition is:

$$B^{N-1} > b^n$$

This may be reformulated as:

$$N > \frac{n}{\log_b B} + 1$$

and if N is to be minimized we therefore have:

$$N = 2 + \lfloor n/(\log_b B) \rfloor$$

No more than this number of digits need ever be output, but there may be some numbers that require exactly that many.

As an example, in converting a binary floating-point number to decimal on the PDP-10 [DEC73] we have $b = 2$, $B = 10$, and $n = 27$. One might naïvely suppose that 27 bits equals 9 octal digits, and if nine octal digits suffice surely 9 decimal digits should suffice also. However, using Matula's formula above, we find that to print such a number may require $2 + \lfloor 27/(\log_2 10) \rfloor = 2 + \lfloor 27/(3.3219\ldots) \rfloor = 2 + \lfloor 8.1278\ldots \rfloor = 10$ decimal digits. There are indeed some numbers that require 10 digits, such as the one which is greater than the rounded 27-bit floating-point approximation to 0.1 by two units in the last place. On the PDP-10 this is the number represented in octal as 175631463150_8, which represents the value $.63146315_8 \times 8^{-1}$. It is intuitively easy to see why ten digits might be needed: splitting the number into an integer part (which is zero) and a fractional part must shift three zero bits into the significand from the left, producing a 30-bit fraction, and 9 decimal digits is not enough to represent all 30-bit fractions.

The condition specified above assumes that the conversions in both directions round correctly. There are other possibilities, such as using truncation (always rounding towards zero) instead of true rounding. If the input conversion rounds but the output conversion truncates, the internal identity requirement can still be met, but more external digits may be required; the condition is $B^{N-1} > 2 \times b^n - 1$. If conversion truncates in both directions, however, it is possible that the internal identity requirement cannot be met, and that repeated conversions may cause a value to "drift" very far from its original value. This implies that if a data base is maintained in decimal, and is updated by converting it to binary, processing it, and then converting back to decimal, values not updated may nevertheless be changed, after many iterations, by a substantial amount [Matula70]. That is why we assume and require proper rounding. Rounding can guarantee the integrity of the data (in the form of the internal identity requirement) and also will minimize the number of output digits. Truncation cannot and will not.

The external identity requirement obviously cannot be strictly satisfied, because by assumption there are a finite number of internally representable values and an infinite number of external values. For every internal value there are many corresponding external representations. From among the several external representations that correspond to a given internal representation, we prefer to get one that is closest in value but also as short as possible.

Satisfaction of the internal identity requirement implies that an *external consistency requirement* be met: if an internal value is printed out, then reading that exact same external value and then printing it once more will produce the same external representation. This is important to the user of an interactive system. If one types `PRINT 3.1415926535` and the system replies `3.1415926`, the user might think "Well and good; the computer has truncated it, and that's all I need type from now on." The user would be rightly upset to then type in `PRINT 3.1415926` and receive the response `3.1415925`! Many language implementations indeed exhibit such undesirable "drifting" behavior.

Let us formulate our desires. First, in the interest of brevity and readability, conversion to external form should produce as few digits as possible. Now suppose, for some internal number, that there are several external representations that can be used: all have the same number of digits. all can be used to reconstruct the internal value precisely, and no shorter external number suffices for the reconstruction. In this case we prefer the one that is closest to the value of the binary number. (As we shall see, all the external numbers must be alike except in the last digit; satisfying this criterion therefore involves only the correct choice for the last digit to be output.)

On the other hand, suppose that the number of digits to be output has been predetermined (fixed-format output); then we desire that the printed value be as close as possible to the binary value. In short, the external form should always be correctly rounded. (The Fortran 77 standard *requires* that floating-point output be correctly rounded [ANSI76, 13.9.5.2]; however, many Fortran implementations do round not correctly, especially for numbers of very large or very small magnitude.)

We first present and prove an algorithm for printing fixed-point fractions that has four useful properties:

(a) *Information preservation.* No information is lost in the conversion. If we know the length n of the original radix-b fraction, we can recover the original fraction from the radix-B output by converting back to radix-b and rounding to n digits.

(b) *Minimum-length output.* No more digits than necessary are output to achieve (a). (This implies that the last digit printed is non-zero.)

(c) *Correct rounding.* The output is correctly rounded; that is, the output generated is the closest approximation to the original fraction among all radix-B fractions of the same length, or is one of two closest approximations, and in the latter case it is easy to choose either of the two by any arbitrary criterion.

(d) *Left-to-right generation.* It is never necessary to propagate carries. Once a digit has been generated, it is the correct one.

These properties are characterized still more rigorously in the description of the algorithm itself.

From this algorithm we then derive a similar method for printing integers. The two methods are then combined to produce two algorithms for printing floating-point numbers, one for free-format applications and one for fixed-format applications.

3 Fixed-Point Fraction Output

Following [Knuth68], we use the notation $\lfloor x \rfloor$ to mean the greatest integer not greater than x. Thus $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -4$. If x is an integer, then $\lfloor x \rfloor = x$. Similarly, we use the notation $\lceil x \rceil$ to mean the smallest integer not less than x; $\lceil x \rceil \equiv -\lfloor -x \rfloor$. Also following [Knuth68], we use the notation $\{x\}$ to mean $x \bmod 1$, or $x - \lfloor x \rfloor$, the fractional part of x. We always indicate numerical multiplication of a and b explicitly as $a \times b$, because we use simple juxtaposition to indicate digits in a place-value notation.

The idea behind the algorithm is very simple. The potential error in a fraction of limited precision may be described as being equal to one-half the minimal non-zero value of the least significant digit of the fraction. The algorithm merely initializes a variable M to this value to represent the precision of the fraction. Whenever the fraction is multiplied by B, the error M is also multiplied by B. Therefore M always has a value against which the residual fraction may be meaningfully compared to decide whether the precision has been exhausted.

Algorithm (FP)[3]:
Finite-Precision Fixed-Point Fraction Printout[1]

Given:

- An n-digit radix-b fraction f, $0 \le f < 1$:

$$f = .f_{-1}f_{-2}f_{-3}\cdots f_{-n} = \sum_{i=1}^{n} f_{-i} \times b^{-i}$$

- A radix B (an integer greater than 1).

Output: The digits F_i of an N-digit ($N \ge 1$) radix-B fraction F:

$$F = .F_{-1}F_{-2}F_{-3}\ldots F_{-N} = \sum_{i=1}^{N} F_{-i} \times B^{-i}$$

such that:

(a) $|F - f| < \dfrac{b^{-n}}{2}$

(b) N is the smallest integer ≥ 1 such that (a) can be true.

(c) $|F - f| \le \dfrac{B^{-N}}{2}$

(d) Each digit is output before the next is generated; it is never necessary to "back up" for corrections

Procedure: see Table 1. ∎

The algorithmic procedure is expressed in pseudo-ALGOL. We have used the **loop-while-repeat** ("$n+\frac{1}{2}$ loop") construct credited to Dahl [Knuth74]. The loop is exited from the **while**-point if the **while**-condition is ever false; thus one may read "**while** B:" as "**if not** B **then exitloop fi**". The **cases** statement is to be taken as a guarded **if** construction [Dijkstra76]; the branch labeled by the true condition is executed, and if both conditions are true (here, if $R = \frac{1}{2}$) either branch may be chosen. The ambiguity in the **cases** statement here reflects the point of ambiguity when

[1] It is difficult to give short, meaningful names to each of a group of algorithms that are similar. Here we give each algorithm a long name that has enough adjectives and other qualifiers to distinguish it from the others, but these long names are unwieldy. For convenience, and as a bit of a joke, two kinds of abbreviated names are used here. Algorithms that are intermediate versions along a path of development have names such as "$(FP)^3$" that are effectively contracted acronyms for the long names. Algorithms that are in "final form" have names of the form "Dragon$k$" for integers k; these algorithms have long names whose acronyms form sequences of letters "F" and "P" that specify the shape of so-called "dragon curves" [Gardner77].

Table 1: Procedure for Finite-Precision Fixed-Point Fraction Printout ((FP)^3)

```
begin
  k ← 0;
  R ← f;
  M ← b^{-n}/2;
  loop
    k ← k + 1;
    U ← ⌊R × B⌋;
    R ← {R × B};
    M ← M × B;
   while R ≥ M and R ≤ 1 − M :
    F_{-k} ← U;
  repeat;
  cases
    R ≤ 1/2 : F_{-k} ← U;
    R ≥ 1/2 : F_{-k} ← U + 1;
  endcases;
  N ← k;
end
```

two radix-B representations of length n are equidistant from f. A given implementation may use any decision method when $R = \frac{1}{2}$, such as "always U", which effectively means to round down; "always $U+1$", meaning to round up; or "if $U \equiv 0 \pmod 2$ then U else $U+1$", meaning to round so that the last digit is even. Note, however, that using "always U" does not *always* round down if rounding up will allow one fewer digit to be printed.

This method is a generalization of the one presented by Taranto [Taranto59] and mentioned in exercise 4.4.3 of [Knuth69].[2]

4 Proof of Algorithm (FP)^3

(The reader who is more interested in practical applications than in details of proof may wish to skip this section.)

In order to demonstrate that Algorithm (FP)^3 satisfies the four claimed properties (a)–(d), it is useful first to prove, by induction on k, the following invariants true at the top of the loop body:

$$M_k = \frac{b^{-n} \times B^k}{2} \qquad (i)$$

$$R_k \times B^{-k} + \sum_{i=1}^{k} F_{-i} \times B^{-i} = f \qquad (ii)$$

By M_k and R_k we mean the values of M and R at the top of the loop as a function of k (the values of M and R change if and only if k is incremented). Invariant (i) is easily verified; we shall take it for granted. Invariant (ii) is a little more complicated.

Basis. After the first two assignments in the procedure, $k = 0$ and $R_0 = f$. The summation in (ii) has no summands and is therefore zero, yielding

$$R_0 \times B^{-0} + 0 = f$$

which is true, because initially $R = f$.

Induction. Suppose (ii) is true for k. We note that the only path backward in the loop sets $F_{-(k+1)} \leftarrow \lfloor R_k \times B \rfloor$. It follows that:

$$\begin{aligned}
f &= R_k \times B^{-k} + \sum_{i=1}^{k} F_{-i} \times B^{-i} \\
&= (B \times R_k) \times B^{-(k+1)} + \sum_{i=1}^{k} F_{-i} \times B^{-i} \\
&= (\{R_k \times B\} + \lfloor R_k \times B \rfloor) \times B^{-(k+1)} + \sum_{i=1}^{k} F_{-i} \times B^{-i} \\
&= (R_{k+1} + F_{-(k+1)}) \times B^{-(k+1)} + \sum_{i=1}^{k} F_{-i} \times B^{-i} \\
&= R_{k+1} \times B^{-(k+1)} + \sum_{i=1}^{k+1} F_{-i} \times B^{-i}
\end{aligned}$$

which establishes the desired invariant.

The procedure definitely terminates, because M initially has a strictly positive value and is multiplied by B (which is greater than 1) each time through the loop. Eventually we have $M > \frac{1}{2}$, at which point $R \geq M$ and $R \leq 1 - M$ cannot both be true and the loop must terminate. (Of course, the loop may terminate before $M > \frac{1}{2}$, depending on R.)

When the procedure terminates, there may be one of two cases, depending on R_N. Because of the rounding step, we have one of the following:

$$f = F + R_N \times B^{-N} \qquad \text{if } R_N \leq \tfrac{1}{2}$$

$$f = F - (1 - R_N) \times B^{-N} \qquad \text{if } R_N \geq \tfrac{1}{2}$$

Let us define R^* as follows:

$$R^* = R_N \qquad \text{if } R_N \leq \tfrac{1}{2}$$

$$R^* = 1 - R_N \qquad \text{if } R_N \geq \tfrac{1}{2}$$

Because $0 \leq R_n \leq 1$, we know that $0 \leq R^* \leq \frac{1}{2}$; we can therefore write:

[2] By the way, the paper that was forward-referenced in the answer to exercise 4.4.3 in [Knuth81] was an early draft of this paper that contained only Algorithm (FP)^3, its proof, and a not-yet-correct generalization to floating-point numbers.

$$f - F = R^* \times B^{-N} \qquad \text{if } R_N \leq \tfrac{1}{2}$$

$$F - f = R^* \times B^{-N} \qquad \text{if } R_N \geq \tfrac{1}{2}$$

From this we have:

$$|F - f| = R^* \times B^{-N} \leq \frac{B^{-N}}{2}$$

proving property (c) (*correct rounding*).

To prove property (a) (*information preservation*), we consider the two cases $M_N > \frac{1}{2}$ and $M_N \leq \frac{1}{2}$. If we have $M_N > \frac{1}{2}$, then

$$|F - f| \leq \frac{B^{-N}}{2} < M_N \times B^{-N} = \frac{b^{-n}}{2}$$

because $M_N = \dfrac{B^N \times b^{-n}}{2}$. If we have $M_N \leq \frac{1}{2}$, then with the fact that $R_N < M_N$ or $R_N > 1 - M_N$ we have $R^* < M_N$, and so

$$|F - f| = R^* \times B^{-N} < M_N \times B^{-N} = \frac{b^{-n}}{2}$$

Property (a) therefore holds in all cases.

To prove property (b) (*minimum-length output*), let us suppose that, to the contrary, there is some P-digit radix-B fraction G such that:

$$P < N \qquad \text{and} \qquad |G - f| < \frac{b^{-n}}{2}$$

In fact, we assume $P = N - 1$ without loss of generality by allowing G to have trailing zero digits to fill out to P places. Now from invariant (*ii*) we know that:

$$R_P \times B^{-P} + \sum_{i=1}^{P} F_{-i} \times B^{-i} = f$$

It follows easily, because B and the F_{-i} are integers and $0 \leq R_P < 1$, that the closest P-digit representations to f are

$$G_1 = .F_{-1}F_{-2}F_{-3}\ldots F_{-(P-1)}F_{-P}$$

and

$$G_2 = .F_{-1}F_{-2}F_{-3}\ldots F_{-(P-1)}(F_{-P} + 1).$$

But because the iteration in the algorithm did not terminate we know that:

$$R_P \geq M_P = \frac{b^{-n} \times B^P}{2}$$

and

$$R_P \leq (1 - M_P).$$

It follows that whether G is chosen to be G_1 or G_2 (and any other choice is even worse) that:

$$|G - f| \geq M_P \times B^{-P} = \frac{b^{-n}}{2}$$

producing a contradiction and proving property (b).

Property (d) (*left-to-right generation*) follows immediately, because the only digit that is ever corrected (by adding one) is the last one output. If that correction were to produce a carry (i.e., $U = B - 1$, and so $U + 1 = B$) then it would mean that the N-digit radix-B output would end in a zero digit, implying that a shorter radix-B fraction could have satisfied property (a); but this is not possible by property (b).

The algorithms that follow are all variants of the one just proven. No other proofs are presented below; the necessary ideas for proving the remaining algorithms are simple modifications of those in the proof just given. However, the code given for later algorithms contains assertions of important invariants; from these invariants complete proofs may be reconstructed.

5 Some Observations

The same routine can be used for converting fractions of various lengths (handling, for example, both single and double precision) provided that the radix-b arithmetic is sufficiently accurate for the longest. The only part of the algorithm dependent on n is the initialization of M.

All of the arithmetic is easy to perform in radix b; the only operations used are multiplication, integer and fractional part, subtraction from 1, and comparison. There is a difficulty if the radix b is odd, because the initialization of M requires division by 2; if the problem should arise, one can either do the arithmetic in an even radix b' that is a multiple of b, or initialize M to b^{-n} (instead of $b^{-n}/2$) and change the comparison $R \geq M \wedge R \leq 1 - M$ to $2 \times R \geq M \wedge 2 \times R \leq 2 - M$.

The accuracy required for the arithmetic is $n + 1 + \lceil \log_b(B-1) \rceil$ radix-b digits: n is the length of the fraction (and of all remainders), 1 more is needed for the factor of $\frac{1}{2}$ in M, and $\lceil \log_b(B-1) \rceil$ more are needed for the result of the multiplication to produce a radix-B digit in radix b.

Algorithm (FP)^3 is (almost) suitable for output of floating-point numbers; it can be used if the floating-point number is first scaled properly.

Algorithm Dragon2:
Floating-Point Printout

Given:

- A radix-b floating-point number $v = f \times b^{(e-p)}$, where e, f, and p are integers such that $p \geq 0$ and $0 \leq f < b^p$.
- A radix B (an integer greater than 1).

Output: A radix-B floating-point approximation to Q, using exponential notation if Q is very large or very small.

Procedure: If $f = 0$ then simply print "0.0". Otherwise proceed as follows. First compute $v' = v \otimes B^{-x}$, where "$\otimes$" denotes floating-point multiplication and where x is chosen so as to make $v' < b^p$. (If exponential format is to be used, one normally chooses x so that the result either is between $1/B$ and 1 or is between 1 and B.) Next, print the integer part of v' by any suitable method, such as the usual division-remainder technique; after that print a decimal point. Then take the fractional part f of v', let $n = p - \lfloor \log_b v' \rfloor - 1$, and apply Algorithm (FP)^3 to f and n. Finally, if x was not zero, the letter "E" and a representation of the scale factor x are printed. ∎

We note that if a floating-point number is the same size (in bits, or radix-b digits, or whatever) as an integer, then arithmetic on integers of that size is often sufficiently accurate, because the bits used for the floating-point exponent are usually more than enough for the extra digits of precision required.

This method for floating-point output is not entirely accurate, but is very simple to code, typically requiring only single-precision integer arithmetic. For applications where producing pleasant output is important, program and data space must be minimized, and the internal identity requirement may be relaxed, this method is excellent. An example of such an application might be an implementation of BASIC for a microcomputer, where pleasant output and memory conservation are more important than absolutely guaranteed accuracy. [The last sentence was written circa 1981. Nowadays all but the tiniest computers have the memory space for the full algorithm.] This algorithm was used for many years in the MacLisp interpreter and found to be satisfactory for most purposes.

There are two problems that prevent Algorithm Dragon2 from being completely accurate for floating-point output.

The first problem is that the spacing between adjacent floating-point numbers of a given precision is not uniform. Let v be a floating-point number, and let v^- and v^+ be respectively the next-lower and next-higher floating-point numbers of the same precision. For most values of v, we have $v^+ - v = v - v^-$. However, if v is an integral power of b, then instead $v^+ - v = b \times (v - v^-)$; that is, the gap below v is smaller than the gap above v by a factor of b.

The second problem is that the scaling may cause loss of information. There may be two numbers, very close together in value, which when scaled by the same power of B result in the same scaled number, because of rounding. The problem is inherent in the floating-point representation. See [Matula68] for further discussion of this phenomenon.

Table 2: Procedure for Indefinite Precision Integer Printout ((IP)^2)

begin
  $k \leftarrow 0$;
  $R \leftarrow d$;
  $M \leftarrow b^n$;
  $S \leftarrow 1$;
  loop
    $S \leftarrow S \times B$;
    $k \leftarrow k + 1$;
   while $(2 \times R) + M \geq 2 \times S$:
  repeat;
  $H \leftarrow k - 1$;
  assert $S = B^{(H+1)}$
  loop
    $k \leftarrow k - 1$;
    $S \leftarrow S/B$;
    $U \leftarrow \lfloor R/S \rfloor$;
    $R \leftarrow R \bmod S$;
   while $2 \times R \geq M$ **and** $2 \times R \leq (2 \times S) - M$:
    $D_k \leftarrow U$;
  repeat;
  cases
    $2 \times R \leq S$: $D_k \leftarrow U$;
    $2 \times R \geq S$: $D_k \leftarrow U + 1$;
  endcases;
  $N \leftarrow k$;
end

6 Conversion of Integers

To avoid the round-off errors introduced by scaling floating-point numbers, we develop a completely accurate floating-point output routine that performs no floating-point computations. As a first step, we exhibit an algorithm for printing *integers* that has a termination criterion similar to that of Algorithm (FP)^3.

Algorithm (IP)^2:
Indefinite Precision Integer Printout
Given:

- An h-digit radix-b integer d, accurate to position n ($n \geq 0$):

$$d = d_h d_{h-1} \ldots d_{n+1} d_n \langle n \text{ zeros} \rangle . = \sum_{i=n}^{h} d_i \times b^i$$

- A radix B.

Output: Integers H and N ($H \geq N \geq 1$) as an H-digit radix-B integer D:

$$D = D_H D_{H-1} \ldots D_{N+1} D_N \langle N \text{ zeros} \rangle. = \sum_{i=N}^{H} D_i \times B^i$$

such that:

(a) $|D - d| < \frac{b^n}{2}$

(b) N is the largest integer, and H the smallest, such that (a) can be true.

(c) $|D - d| \leq \frac{B^N}{2}$

(d) Each digit is output before the next is generated.

Procedure: see Table 2. ∎

In Algorithm $(\text{IP})^2$ all quantities are integers. We avoid the use of fractional quantities by introducing a scaling factor S and logically replacing the quantities R and M by R/S and M/S. Whereas in Algorithm $(\text{FP})^3$ the variables R and M were repeatedly multiplied by B, in Algorithm $(\text{IP})^2$ the variable S is repeatedly divided by B.

7 Free-Format Floating-Point Output

From these two algorithms, $(\text{FP})^3$ and $(\text{IP})^2$, it is now easy to synthesize an algorithm for "perfect" conversion of a (positive) fixed-precision floating-point number to a free-format floating-point number in another radix. We shall assume that a floating-point number f is represented as a tuple of three integers: the exponent e, the "fraction" or "significand" m, and the precision p. Together these integers represent the mathematical value $f = m \times b^{(e-p)}$. We require $m < b^p$; a normalized representation additionally requires $m \geq b^{(p-1)}$, but we do not depend on this.

We define the function $shift_b$ of two integer arguments x and n ($x \geq 0$):

$$shift_b(x, n) \equiv \lfloor x \times b^n \rfloor$$

This function is intended to be trivial to implement in radix-b arithmetic: it is the result of shifting x by n radix-b digit positions, to the left for positive n (introducing low-order zeros) or to the right for negative n (discarding digits shifted to the right of the "radix-b point").

Algorithm $(\text{FPP})^2$:

Fixed-Precision Positive Floating-Point Printout

Given:

- Three radix-b integers e, f, and p, representing the number $v = f \times b^{(e-p)}$, with $p \geq 0$ and $0 < f < b^p$,
- A radix B.

Output: Integers H and N and digits D_k ($H \geq k \geq N$) such that if one defines the value

$$V = \sum_{i=N}^{H} D_i \times B^i$$

then:

(a) $\frac{v^- + v}{2} < V < \frac{v + v^+}{2}$

(b) H is the smallest integer (that is, furthest from ∞) and N the largest integer (that is, furthest from $-\infty$) such that (a) can be true

(c) $|V - v| \leq \frac{B^N}{2}$

(d) Each digit is output before the next is generated.

Procedure: see Tables 3 and 4. ∎

Unfortunately, this algorithm requires $f \neq 0$; it does not work properly for zero. Notice that relationship (a) is not stated in the symmetric form $|V - v| < x$. The relationship is not symmetrical because of the phenomenon of unequal gaps.

As in algorithm $(\text{IP})^2$, all arithmetic operations produce integer results and all variables take on integer values. This is done by using the scale factor $S = shift_b(1, -(e-p)) = \lfloor b^{(p-e)} \rfloor$. Note, however, that some of these integer values may be *very* large. If this algorithm is used to print (in decimal) floating-point numbers in IEEE standard single-precision floating-point format [IEEE81, IEEE85], integers as large as 2^{154} may be calculated; for double-precision format integers as large as 2^{1050} may be encountered. Such a large integer can be represented in fewer than five 32-bit words (for single precision) or forty 32-bit words (for double precision). While multi-precision integer arithmetic is required in the general case the storage requirements are certainly not impractical. The multiprecision arithmetic does not have to be quite as general as that presented in [Knuth69, §4.3.1]. One must add, subtract, and compare multiprecision integers; multiply and divide multiprecision integers by B; and divide one multiprecision integer by another (but only in situations where it is known that the result will be a radix-B digit, which is much simpler than the general case). The size of the integers involved depends primarily on the exponent value of the floating-point number to be printed. For floating-point numbers of reasonable magnitude, 32-bit or 64-bit integer arithmetic is likely to suffice and so execution speed is typically not a problem.

The successor v^+ and predecessor v^- to a positive floating-point number $v = f \times b^{(e-p)}$ are taken to be

$$v^+ = v + b^{(e-p)}$$

and

$$v^- = \textbf{if } f = b^{(p-1)} \textbf{ then } v - b^{(e-p-1)} \textbf{ else } v - b^{(e-p)} \textbf{ fi}$$

Table 3: Fixed-Precision Positive Floating-Point Printout ((FPP)²)

begin
  assert $f \neq 0$
  $R \leftarrow shift_b(f, \max(e-p, 0))$;
  $S \leftarrow shift_b(1, \max(0, -(e-p)))$;
  assert $R/S = f \times b^{(e-p)} = v$
  $M^- \leftarrow shift_b(1, \max(e-p, 0))$;
  $M^+ \leftarrow M^-$;
  Simple-Fixup;
  $H \leftarrow k-1$;
  loop
    assert $(R/S) \times B^k + \sum_{i=k}^{H} D_i \times B^i = v$
    $k \leftarrow k-1$;
    $U \leftarrow \lfloor (R \times B)/S \rfloor$;
    $R \leftarrow (R \times B) \bmod S$;
    $M^- \leftarrow M^- \times B$;
    $M^+ \leftarrow M^+ \times B$;
    $low \leftarrow 2 \times R < M^-$;
    $high \leftarrow 2 \times R > (2 \times S) - M^+$;
  while (not *low***) and (not** *high***) :**
    $D_k \leftarrow U$;
  repeat;
  comment Let $V_k = \sum_{i=k+1}^{H} D_i \times B^i$.
  assert $low \Rightarrow \frac{v^- + v}{2} < (U \times B^k + V_k) \leq v$
  assert $high \Rightarrow v \leq ((U+1) \times B^k + V_k) < \frac{v^+ + v}{2}$
  cases
    low **and not** *high* : $D_k \leftarrow U$;
    high **and not** *low* : $D_k \leftarrow U+1$;
    low **and** *high* :
      cases
        $2 \times R \leq S$: $D_k \leftarrow U$;
        $2 \times R \geq S$: $D_k \leftarrow U+1$;
      endcases;
  endcases;
  $N \leftarrow k$;
end;

Table 4: Procedure *Simple-Fixup*

procedure *Simple-Fixup*;
  begin
    assert $R/S = v$
    assert $M^-/S = M^+/S = b^{(e-p)}$
    if $f = shift_b(1, p-1)$ **then**
      $M^+ \leftarrow shift_b(M^+, 1)$;
      $R \leftarrow shift_b(R, 1)$;
      $S \leftarrow shift_b(S, 1)$;
    fi;
    $k \leftarrow 0$;
    loop
      assert $(R/S) \times B^k = v$
      assert $(M^-/S) \times B^k = v - v^-$
      assert $(M^+/S) \times B^k = v^+ - v$
    while $R < \lceil S/B \rceil$:
      $k \leftarrow k-1$;
      $R \leftarrow R \times B$;
      $M^- \leftarrow M^- \times B$;
      $M^+ \leftarrow M^+ \times B$;
    repeat;
    assert $k = \min(0, 1 + \lfloor \log_B v \rfloor)$
    loop
      assert $(R/S) \times B^k = v$
      assert $(M^-/S) \times B^k = v - v^-$
      assert $(M^+/S) \times B^k = v^+ - v$
    while $(2 \times R) + M^+ \geq 2 \times S$:
      $S \leftarrow S \times B$;
      $k \leftarrow k+1$;
    repeat;
    assert $k = 1 + \left\lfloor \log_B \frac{v + v^+}{2} \right\rfloor$
  end;

The formula for v^+ does not have to be conditional, even when the representation for v^+ requires a larger exponent e than v does in order to satisfy $f < b^p$. The formula for v^-, however, must take into account the situation where $f = b^{(p-1)}$, because v^- may then use the next-smaller value for e, and so the gap size is smaller by a factor of b. There may be some question as to whether this is the correct criterion for floating-point numbers that are not normalized. Observe, however, that this is the correct criterion for IEEE standard floating-point format [IEEE85], because all such numbers are normalized except those with the smallest possible exponent, so if v is denormalized then v^- must also be denormalized and have the same value for the exponent e.

To compensate for the phenomenon of unequal gaps, the variables M^- and M^+ are given the value that M would have in Algorithm (IP)^2, and then adjustments are made to M^+, R, and S if necessary. The simplest way to account for unequal gaps would be to divide M^- by b. However, this might cause M^- to have a non-integral value in some situations. To avoid this, we instead scale the entire algorithm by an additional factor of b, by multiplying M^+, R, and S by b.

The presence of unequal gaps also induces an asymmetry in the termination test that reveals a previously hidden problem. In Algorithm (IP)^2, with $M^- = M^+ = M$, it was the case that if $R > (2 \times S) - M^+$ and $2 \times R < S$, then necessarily $R < M^-$ also. Here this is not necessarily so, because M^- may be smaller than M^+ by a factor of b. The interpretation of this is that in some situations one may have $2 \times R < S$ but nevertheless must round up to the digit $U + 1$, because the termination test succeeds relative to M^+ but fails relative to M^-.

This problem is solved by rewriting the termination test into two parts. The boolean variable *low* is true if the loop may be exited because M^- is satisfied (in which case the digit U may be used). The boolean variable *high* is true if the loop may be exited because M^+ is satisfied (in which case the digit $U + 1$ may be used). If both variables are true, then either digit of U and $U + 1$ may be used, as far as information-preservation is concerned; in this case, and this case only, the comparison of $2 \times R$ to S should be done to satisfy the correct-rounding property.

Algorithm (FPP)^2 is conceptually divided into four parts. First, the variables R, S, M^-, and M^+ are initialized. Second, if the gaps are unequal then the necessary adjustment is made. Third, the radix-B weight H of the first digit is determined by two loops. The first loop is needed for positive H, and repeatedly multiplies S by B; the second loop is needed for negative H, and repeatedly multiplies R, M^-, and M^+ by B. (Divisions by B are of course avoided to prevent roundoff errors.) Fourth, the digits are generated by the last loop; this loop should be compared with the one in Table 1.

8 Fixed-Format Floating-Point Output

Algorithm (FPP)^2 simply generates digits, stopping only after producing the appropriate number of digits for precise free-format output. It needs more flexible means of cutting off digit generation for fixed-format applications. Moreover, it is desirable to intersperse formatting with digit generation rather than buffering digits and then re-traversing the buffer.

It is not difficult to adapt Algorithm (FPP)^2 for fixed-format output, where the number of digits to be output is predetermined independently of the floating-point value to be printed. Using this algorithm avoids giving the illusion of more internal precision than is actually present, because it cuts off the conversion after sufficiently many digits have been produced; the remaining character positions are then padded with blanks or zeros.

As an example, suppose that a Fortran program is written to calculate π, and that it indeed calculates a floating-point number that is the best possible approximation to π for that floating-point format, precise to, say, 27 bits. However, it then prints the value with format `F20.18`, producing the output "`3.14159265160560607 9`".

This is indeed accurately the value of that floating-point number to that many places, but the last ten digits are in some sense not justified, because the internal representation is not that precise. Moreover, this output certainly does not represent the value of π accurately; in format `F20.18`, π should be printed as "`3.14159265358979323 8`". The free-format printing procedure described above would cut off the conversion after nine digits, producing "`3.14159265`" and no more. The fixed-format procedure to be developed below would therefore produce "`3.14159265 `" or "`3.14159265000000000`". Either of these is much less misleading as to internal precision.

The fixed-format algorithm works essentially uses the free-format output method, differing only in when conversion is cut off. If conversion is cut off by exhaustion of precision, then any remaining character positions are padded with blanks or zeros. If conversion is cut off by the format specification, the only problem is producing correctly rounded output. This is easily *almost*-solved by noting that the remainder R can correctly direct rounding at *any* digit position, not just the last. In terms of the programs, *almost* all that is necessary is to terminate a conversion loop prematurely if appropriate, and the following cases statement will properly round the last digit produced.

This doesn't solve the entire problem, however, because if conversion is cut off by the format specification then the early rounding may require propagation of carries; that is, the proof of property (d) fails to hold. This can be taken care of by adjusting the values of M^- and M^+ appropriately. In principle this adjustment sometimes requires the use of non-integral values that cannot be represented exactly in radix b; in practice such values can be rounded to get the proper effect.

As noted above, it is indeed not difficult to adapt the simplified algorithm for a *particular* fixed format. However, straightforwardly adapting it to handle a *variety* of fixed formats is clumsy. We tried to do this, and found that we had to introduce many switches and flags to control the various aspects of formatting, such as total field width, number of digits before and after the decimal point, width of exponent field, scale factor (as for "P" format specifiers in Fortran), and so on. The result was a tangled mess, very difficult to understand and maintain.

We finally untangled the mess by completely separating the *generation* of digits from the *formatting* of the digits. We added a few interface parameters to produce a digit generator (called Dragon4) to which a variety of formatting processes could be easily interfaced. The generator and the formatter execute as coroutines, and it is assumed that the "user" process executes as a third coroutine.

For purposes of exposition we have borrowed certain features of Hoare's CSP (Communicating Sequential Processes) notation [Hoare78]. The statement

$$\text{GENERATE!}\,(x_1, \ldots, x_n)$$

means that a message containing values $x_1, \ldots, x_n$ is to be sent to the GENERATE process; the process that executes such a statement then continues its own execution without waiting for a reply. The statement

$$\text{FORMAT?}\,(x_1, \ldots, x_n)$$

means that a message containing values $x_1, \ldots, x_n$ is to be received from the FORMAT process; the process that executes such a statement waits if necessary for a message to arrive. (The notation is symmetrical in that the sender and receiver of a message must each know the other's name.) We do *not* borrow any of the control structure notations of CSP, and we do not worry about the matter of processes failing.

To perform floating-point output one must effectively execute three coroutines together. In the CSP notation one writes:

```
[ USER :: user process |
  FORMAT :: formatting process |
  GENERATE :: Dragon4 ]
```

The interface convention is that the "user process" must first send the message

$$\text{FORMAT!}\,(b, e, f, p, B)$$

to initiate floating-point conversion and then may receive characters from the FORMAT process until a "@" character is received. The "@" character serves as a terminator and should be discarded. Any of the formatting processes shown below may be used; to get free-format conversion, for example, one would execute

```
[ USER :: user process |
  FORMAT :: Free-Format |
  GENERATE :: Dragon4 ]
```

and to get fixed-format exponential formatting one would execute

```
[ USER :: user process |
  FORMAT :: Fixed-Format Exponential |
  GENERATE :: Dragon4 ]
```

We have found it easy to write new formatting routines.

9 Implementations

In 1981 we coded a version of Algorithm Dragon4 in Pascal, including the formatting routines shown here as well as formatters for Fortran E, F, and G formats and for PL/I-style picture formats such as `$$$,$$$,$$9.99CR`. This suite has been tested thoroughly. This algorithm has also been used in various Lisp implementations for both free-format and fixed-format floating-point output. A portable and performance-tuned implementation in C is in progress.

10 Historical Note and Acknowledgments

The work reported here was done in the late 1970's. This is the first published appearance, but earlier versions of this paper have been circulated "unpublished" through the grapevine for the last decade. The algorithms have been used for years in a variety of language implementations, perhaps most notably in Zetalisp.

Why wasn't it published earlier? It just didn't seem like that big a deal; but we were wrong. Over the years floating-point printers have continued to be inaccurate and we kept getting requests for the unpublished draft. We thank Donald Knuth for giving us that last required push to submit it for publication.

The paper has been almost completely revised for presentation at this conference. An intermediate version of the algorithm, Dragon3 (Free-Format Perfect Positive Floating-Point Printout), has been omitted from this presentation for reasons of space; it is of interest only

in being closest in form to what was actually first implemented in MacLisp. We have allowed a few anachronisms to remain in this presentation, such as the suggestion that a tiny version of the printing algorithm might be required for microcomputer implementations of BASIC. You will find that most of the references date back to the 1960's and 1970's.

Helpful and valuable comments on drafts of this paper were provided by Jon Bentley, Barbara K. Steele, and Daniel Weinreb. We are also grateful to Donald Knuth and Will Clinger for their encouragement.

The first part of this work was done in the mid to late 1970's while both authors were at the Massachusetts Institute of Technology, in the Laboratory for Computer Science and the Artificial Intelligence Laboratory. From 1978 to 1980, Steele was supported by a Fanny and John Hertz Fellowship.

This work was carried forward by Steele at Carnegie-Mellon University, Tartan Laboratories, and Thinking Machines Corporation, and by White at IBM T. J. Watson Research and Lucid, Inc. We thank these institutions for their support.

This work was supported in part by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

11 References

[ANSI76] American National Standards Institute. *Draft proposed ANS Fortran (BSR X3.9).* Reprinted as *ACM SIGPLAN Notices* 11, 3 (March 1976).

[Clinger90] Clinger, William D. How to read floating point numbers accurately. Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, New York, June 1990).

[DEC73] Digital Equipment Corporation. *DecSystem 10 Assembly Language Handbook.* Third edition. (Maynard, Massachusetts, 1973).

[Dijkstra76] Dijkstra, Edsger W. *A Discipline of Programming.* Prentice-Hall (Englewood Cliffs, New Jersey, 1976).

[Gardner77] Gardner, Martin. "The Dragon Curve and Other Problems." In *Mathematical Magic Show.* Knopf (New York, 1977), 203–222.

[Hoare78] Hoare, C. A. R. "Communicating Sequential Processes." *Communications of the ACM* 21, 8 (August 1978), 666–677.

[IEEE81] IEEE Computer Society Standard Committee, Microprocessor Standards Subcommittee, Floating-Point Working Group. "A Proposed Standard for Binary Floating-Point Arithmetic." *Computer* 14, 3 (March 1981), 51–62.

[IEEE85] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985 (New York, 1985).

[Jensen74] Jensen, Kathleen, and Wirth, Niklaus. *PASCAL User Manual and Report.* Second edition. Springer-Verlag (New York, 1974).

[Knuth68] Knuth, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley (Reading, Massachusetts, 1968).

[Knuth69] Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* First edition. Addison-Wesley (Reading, Massachusetts, 1969).

[Knuth74] Knuth, Donald E. "Structured Programming with GO TO Statements." *Computing Surveys* 6, 4 (December 1974).

[Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Second edition. Addison-Wesley (Reading, Massachusetts, 1981).

[Matula68] Matula, David W. "In-and-Out Conversions." *Communications of the ACM* 11, 1 (January 1968), 47–50.

[Matula70] Matula, David W. "A Formalization of Floating-Point Numeric Base Conversion." *IEEE Transactions on Computers* C-19, 8 (August 1970), 681–692.

[Moon74] Moon, David A. *MacLisp Reference Manual, Revision 0.* Massachusetts Institute of Technology, Project MAC (Cambridge, Massachusetts, April 1974).

[Taranto59] Taranto, Donald. "Binary Conversion, with Fixed Decimal Precision, of a Decimal Fraction." *Communications of the ACM* 2, 7 (July 1959), 27.

Table 5: Procedure Dragon4 (Formatter-Feeding Process for Floating-Point Printout, Performing Free-Format Perfect Positive Floating-Point Printout)

```
process Dragon4;
  begin
    FORMAT?(b, e, f, p, B, CutoffMode, CutoffPlace);
    assert CutoffMode = "relative" ⇒ CutoffPlace ≤ 0
    RoundUpFlag ← false;
    if f = 0 then FORMAT!(0, k) else
      R ← shift_b(f, max(e − p, 0));
      S ← shift_b(1, max(0, −(e − p)));
      M⁻ ← shift_b(1, max(e − p, 0));
      M⁺ ← M⁻;
      Fixup;
      loop
        k ← k − 1;
        U ← ⌊(R × B)/S⌋;
        R ← (R × B) mod S;
        M⁻ ← M⁻ × B;
        M⁺ ← M⁺ × B;
        low ← 2 × R < M⁻;
        if RoundUpFlag
          then high ← 2 × R ≥ (2 × S) − M⁺
          else high ← 2 × R > (2 × S) − M⁺ fi;
       while not low and not high
            and k ≠ CutoffPlace :
        FORMAT!(U, k);
      repeat;
      cases
        low and not high : FORMAT!(U, k);
        high and not low : FORMAT!(U + 1, k);
        (low and high) or (not low and not high) :
          cases
            2 × R ≤ S : FORMAT!(U, k);
            2 × R ≥ S : FORMAT!(U + 1, k);
          endcases;
      endcases;
    fi;
    comment Henceforth this process will generate as
      many "−1" digits as the caller desires, along
      with appropriate values of k.
    loop k ← k − 1; FORMAT!(−1, k) repeat;
  end;
```

Table 6: Procedure *Fixup*

```
procedure Fixup;
  begin
    if f = shift_b(1, p − 1) then
      comment Account for unequal gaps.
      M⁺ ← shift_b(M⁺, 1);
      R ← shift_b(R, 1);
      S ← shift_b(S, 1);
    fi;
    k ← 0;
    loop
     while R < ⌈S/B⌉ :
      k ← k − 1;
      R ← R × B;
      M⁻ ← M⁻ × B;
      M⁺ ← M⁺ × B;
    repeat;
    loop
      loop
       while (2 × R) + M⁺ ≥ 2 × S :
        S ← S × B;
        k ← k + 1;
      repeat;
      comment Perform any necessary adjustment
        of M⁻ and M⁺ to take into account the for-
        matting requirements.
      case CutoffMode of
        "normal" : CutoffPlace ← k;
        "absolute" : CutoffAdjust;
        "relative" :
          CutoffPlace ← k + CutoffPlace;
          CutoffAdjust;
      endcase;
     while (2 × R) + M⁺ ≥ 2 × S :
    repeat;
  end;
```

Table 7: Procedure *fill*

```
procedure fill(k, c);
  comment Send k copies of the character c to the
    USER process. No characters are sent if k = 0.
  for i from 1 to k do USER!(c) od;
```

Table 8: Procedure *CutoffAdjust*

```
procedure CutoffAdjust;
  begin
    a ← CutoffPlace − k;
    y ← S;
    cases
      a ≥ 0 : for j ← 1 to a do y ← y × B;
      a ≤ 0 : for j ← 1 to − a do y ← ⌈y/B⌉;
    endcases;
    assert y = ⌈S × B^a⌉
    M⁻ ← max(y, M⁻);
    M⁺ ← max(y, M⁺);
    if M⁺ = y then RoundUpFlag ← true fi;
  end;
```

Table 9: Procedure *DigitChar*

```
procedure DigitChar(U);
  case U of
    comment A digit that is −1 is treated as a zero
      (one that is not significant). Here we print a
      blank for it; fixed Fortran formats might prefer
      a zero.
    −1 : USER! (" ");
    0 : USER! ("0");
    1 : USER! ("1");
    2 : USER! ("2");
    3 : USER! ("3");
    4 : USER! ("4");
    5 : USER! ("5");
    6 : USER! ("6");
    7 : USER! ("7");
    8 : USER! ("8");
    9 : USER! ("9");
    10 : USER! ("A");
    11 : USER! ("B");
    12 : USER! ("C");
    13 : USER! ("D");
    14 : USER! ("E");
    15 : USER! ("F");
  endcase;
```

Table 10: Formatting process for free-format output

```
process Free-Format;
  begin
    USER ? (b, e, f, p, B);
    GENERATE! (b, e, f, p, B, "normal", 0);
    GENERATE ? (U, k);
    if k < 0 then
      USER! ("0");
      USER! (".");
      fill(−k, "0")
    fi;
    loop
      DigitChar(U);
      if k = 0 then USER! (".") fi;
      GENERATE ? (U, k);
     while U ≠ −1 or k ≥ −1 :
    repeat;
    USER! ("@");
  end;
```

Table 11: Formatting process for fixed-format output

```
process Fixed-Format;
  begin
    USER ? (b, e, f, p, B, w, d);
    assert d ≥ 0 ∧ w ≥ max(d + 1, 2)
    c ← w − d − 1;
    GENERATE! (b, e, f, p, B, "absolute", −d);
    GENERATE ? (U, k);
    if k < c then
      if k < 0 then
        if c > 0 then fill(c − 1, " "); USER! ("0") fi;
        USER! (".");
        fill(min(−k, d), "0");
      else fill(c − k − 1, " ") fi;
      loop
       while k ≥ −d :
        DigitChar(U);
        if k = 0 then USER! (".") fi;
        GENERATE ? (U, k);
      repeat;
    else fill(w, "*") fi;
    USER! ("@");
  end;
```

Table 12: Formatting process for free-format exponential output

```
process Free-Format Exponential;
  begin
    USER ? (b, e, f, p, B);
    GENERATE! (b, e, f, p, B, "normal", 0);
    GENERATE ? (U, expt);
    DigitChar(U);
    USER! (".");
    loop
      GENERATE ? (U, k);
     while U ≠ −1 :
      DigitChar(U);
    repeat;
    if k = expt − 1 then USER! ("0") fi;
    USER! ("E");
    if expt < 0 then USER! ("-"); expt ← −expt fi;
    j ← 1;
    loop j ← j × B; while j ≤ expt : repeat;
    loop
      j ← ⌊j/B⌋;
      DigitChar(⌊expt/j⌋);
      expt ← expt mod j;
     while j > 1 :
    repeat;
    USER! ("@");
  end;
```

Table 13: Formatting process for fixed-format exponential output

```
process Fixed-Format Exponential;
  begin
    USER ? (b, e, f, p, B, w, d, x);
    assert d ≥ 0 ∧ x ≥ 1 ∧ w > d + x + 3
    c ← w − d − x − 3;
    GENERATE! (b, e, f, p, B, "relative", −d);
    GENERATE ? (U, expt);
    j ← 1;
    a ← 0;
    loop
      j ← j × B;
      a ← a + 1;
     while j ≤ |expt| :
    repeat;
    if a ≤ x then
      fill(c − 1, " ");
      DigitChar(U);
      USER! (".");
      for q ← 1 to d do
        GENERATE ? (U, k);
        DigitChar(U);
      od;
      USER! ("E");
      if expt < 0 then
        USER! ("-")
       else
        USER! ("+")
      fi;
      expt ← |expt|;
      fill(x − a, "0");
      loop
        j ← ⌊j/B⌋;
        DigitChar(⌊expt/j⌋);
        expt ← expt mod j;
       while j > 1 :
      repeat;
    else fill(w, "*") fi;
    USER! ("@");
  end
```