Continuity and Limits of Programs

Richard Fateman

September 30, 2003

Abstract

In demonstrations of proofs [3] of correctness for programs that are designed to compute mathematical functions we attempted to show that programs have the right properties, namely the same properties as the mathematical functions they are alleged to compute. In the cited paper we were forced to hand-wave (our excuse was the need for brevity) in at least two places, trying to side-step sticky issues. Here we try to address these issues by clarifying two concepts: (a) computational continuity and (b) equality in a domain where floating-point computations can be done to variable (presumably high) precision. We introduce notations p-representable and p-negligible where p denotes precision, and show how this helps in our applications.

1 Introduction

Creating correct floating-point or other mathematical software is difficult, and techniques for assuring the validity of such software ranges from testing to confirm agreement at many (random, or selected) points [1], or looking at the discrete computations involved in a specific hardware model of computation [7] or trying to generalize by analogy to continuous models. ([9]).

Theoretical models of computing with real numbers have become a popular research area, and of this work we particularly point out LFT [2]. This paper surveys previous significant work to address the gap between what we generally talk about (reals) and what we can compute conventionally. The gap between these concepts is filled by an approach which is essentially to compute successive digits (bits) in a representation of an answer such that each bit progressively refines the answer. A redundant representation is necessary (e.g. digits can be negative) to account for situations in which a decision cannot be made immediately (or ever). One argument for the usefulness of such work is, by recent tradition, applications of computational geometry where delicate evaluations of (usually) simple expressions are necessary to avoid geometric degeneracies. In particular one needs to determine the sign of a number which may ever more closely approximate zero: at a given fixed precision one cannot be sure. This is not the only approach: certainly if the input to geometric predicates is of finite precision, infinite computation should not be required [11]. Other on-line computations for transcendental and trigonometric functions are explored by Potts [10].

Although we are generally appreciative of this kind of question and such approaches, we would like to address a somewhat different question: How can we say of a program that it computes a certain continuous mathematical function "in the limit" – that is, as one increases both the precision of the input and of the operations, and (perhaps) runs more iterations – one gets an answer that is as close as one could wish to the correct answer.

One computer algebra system (Mathematica) claims to have arbitrary precision algorithms to evaluate all its elementary and special functions, as well as a kind of interval arithmetic by which it can convey a bound on the actual accuracy of the result. One may have a highly precise but inaccurate number: 3.1423456790123456789 looks *precise* but is not *accurate* if it is supposed to be π. The methods used by Mathematica are not generally open to scrutiny, and according to the current documentation do not work all

the time: that is, they sometimes give answers to lower precision than requested, and some attention must be paid to reformulating the questions. This is in some sense inevitable in that some computations may essentially never halt if you insist on certain accuracy in the result. As an example, consider computing the first few digits of $1 - \sum_{k=1}^{\infty} 9 \times 10^{-k}$. This may seem like a frivolous example, but in fact some computations based on Taylor series expansions require that you (at least) find the first non-zero term in the expansion. It may not be obvious at first that a term is zero until one does some elaborate simplification. Yet this is not enough, generally. Even if a given term *is* zero, it may be an extremely subtle computation to show that *all* the remaining terms are zero as well.

2 Discrete maps

All digital programs map finite sets to finite sets. From this viewpoint, mathematical function computational "continuity" is irrelevant [7]. Such discrete analysis tends to be extremely tedious and very specific, such as proving that for each of the (huge but finite) number of possible IEEE 754 standard double-precision floating-point numbers, a certain sequence of operations provides the correctly rounded value of the sine.

Our discussion must apply to the kinds of high-precision-float arithmetic supported by CAS and various library packages; although one could argue from a finite-universe perspective there are still only a finite number of such numbers, it is unreasonable to consider testing a function on all such values[1].

The bridge to arbitrary precision leads from discrete, to some approximation, to continuous computation.

3 Continuity

Most of what we say informally about programs to compute mathematical functions is dependent on an assumption of continuity. To be specific, we repeat the first example from a recent paper [3] a program to compute $\sin x$. The definition is $S(x) :=$ if $|x| < \epsilon$ then x else $3 \cdot S(x/3) - 4 \cdot S(x/3)^3$.

This program, perhaps surprisingly, for small enough ϵ appears to have many of the properties, approximately, of $\sin x$. What can we say about continuity? Can we *really* sweep under the rug enough of the numerical analysis issues of truncation, round-off, overflow[2], to proceed to a useful analysis? We will concentrate on the issue of trying to "smooth" things over for simplicity of exposition. For example how could we know if something very unexpected happens "in-between" representable numbers? In this context, can we talk about continuity of floating-point computations?

3.1 Calculation with exact rationals

A common facility in computer algebra systems is support for calculations with exact rationals. The discussion of continuity can be short-circuited by saying that we are only computing with rationals (and after all, even floating-point numbers are subsets of rationals). The exact rational domain has the advantage that one can always construct a new rational between any two distinct rationals, and thus there are constructive limits. Computationally this is generally uncomfortable, since the representation size grows rapidly with operations. As an example, letting $\epsilon = 10^-6$, $s(1/10)$ produces a rational number requiring $26,500$ characters to print. Converted to a floating-point number it looks like $0.099833416681499927...$ where the italic digits differ from the correct value for $\sin(1/10)$. We were clearly carrying far more precision than justified by the termination condition of the algorithm.

Exact rationals clearly run out of representational power as soon as one introduces square-root; even if we introduce algebraic numbers we still do not have sine and cosine. Either we use some kind of continued

[1] By contrast, it *is* conceivable to iterate through all single-precision floating-point fractions to test (say) square-root programs.
[2] We will ignore overflow, since it is generally avoidable with high-precision floats.

computation, interval arithmetic, approximation, or a fairly painful exact extension such as suggested by Hur [4].

Will approximation do the job? An interesting example is given in the previous cited paper by Edalat [2]. Consider the recurrence $a_0 := 11/2,\ a_1 :=: 61/11,\ a_n := 111 - (1130 - 3000/a_{n-2})/a_{n-1}$. This converges to 6.0, computed exactly. Here's a sample value:

$$a_{50} = \frac{48501317539984344264750070554064540883 81}{80837009530673407316637349098998355900 1} = 5.9998\cdots.$$

This same convergence to 6.0 is revealed by computing this sequence in 140 decimal digits. However, in *any precision less than about 100 decimal digits of precision*, it looks like it converges to 100.000. This behavior is distressing since one might easily conclude that if the answer is 100.0 uniformly for 16, 30, 60, 90 decimal digits, then that must be the answer! In the face of such (admittedly contrived) examples, one must cautious about "computational" convergence.

Even in this example, enough precision eventually gets the answer right. Where does this precision come from? The next section discusses this.

3.2 A brief explanation of precision and bigfloats

Readers will be familiar with ordinarily "hardware" floating-point arithmetic. Recall that when we try to represent two distinct real numbers that are too close (differing by less than half a unit in the last place (ULP) of the floating-point fraction) their representations are bit-wise equal. They each are mapped to the single number in a (small enough) ULP neighborhood.

So-called arbitrary-precision floating-point or "bigfloat" systems essentially allow one to specify *a priori* another (usually higher) level of precision. These software systems represent numbers in a form that may require many words in the fraction (and typically a substantial overkill in the exponent representation as well). We remind the reader that even these numbers are explicitly stored in a finite computer and are necessarily finite rational numbers. We saw that rational arithmetic increases the representation size as a result of operations; we also saw that the bloat may not be justified by the computation. While the bigfloat precision may be set globally for all operations, some implementations may provide increased precision the in a fashion similar to rational arithmetic: incrementally as justified by operations. That is, a multiplication of two 32 bit numbers can be represented exactly using no more than 64 bits, etc. This is a convenience for various computations, and can save time compared to setting a single global precision used for the duration of a computation by starting with low precision and growing (see [11] for an example and clever implementation that can also, in effect, allow for "holes" in the fraction to save space.). Done carelessly, growing precision at each of a large number of sequential operations leads to a disaster of exponential growth.

The effect of using bigfloats is to populate the neighborhood around every lower-precision number with more high-precision neighbors.

To emphasize a point mentioned earlier, no arbitrary-precision package allows for exact representation of irrational or transcendental numbers. Only some rational numbers, namely the subset of rationals that can be represented with a finite "fraction" in a given radix, are possible. There are methods for representing transcendental or irrational numbers by a kind of indirection. We could use symbols (e.g. π or $\sqrt{31}$) or black-box programs which can generate any number of digits on request [12].

3.3 Continuity with bigfloats

How can we define continuity of a program F, perhaps relating it to an implementation of the mathematical function f?

Consider one common mathematical definition: $f:\ X \to Y$ is continuous if the pre-image of every open set in Y is an open set in X.

If F maps floats X' to floats Y' and is alleged to compute f, we appear to be stuck: Perhaps F is vacuously continuous because there are no open sets in Y' – there are no open sets in the floating-point domain. But then every program from floats to floats is equally and vacuously continuous.

Maybe we can use an epsilon-delta equivalent. The function f is continuous at a point x if for any $\epsilon > 0$ there exists a $\delta > 0$ such that for any y such that $|x - y| < \delta$, $|f(x) - f(y)| < \epsilon$.

Computationally, we cannot choose an $\epsilon > 0$ unless ϵ is at least as large as the "machine epsilon" of our representation. Software arbitrary precision allows us to construct arbitrarily small rational "software" epsilons, but we must, in some sense, continually chase it to smaller values in a limiting process.

Let us call all the numbers in precision p (and some exponent domain that might grow with p) p-representable. For definiteness less us say that p denotes an exponent of base 2. Thus $1 + 2^{-p}$ is different from 1, but $1 + 2^{-p-1}$ is not representable except as 1.

The *program* F is precision-p-continuous at a precision-p-representable point x if for any p-representable $\epsilon > 0$ there exists a $\delta > 0$ (which may require for its representation a higher precision p') for which the following statement holds in p'-precision: $|x - y| < \delta$ implies $|F(x) - F(y)| < \epsilon$.

note This requires that x and y be representable exactly in precision p' and that the program F can compute all of its operations in any required precision. This latter condition is *not* typical of scientific subroutine libraries which are almost universally written to achieve required accuracies based on using specific machine precisions. They typically use algorithms that are as efficient as possible. A good design will be just barely accurate enough to meet the requirements; any computation will cost more, and not be apparent in the answer!

3.4 Failure continuity

Hamlet [9] introduces the notion of *failure continuity* to formalize the notion that if a program fails to meet its specification at some input x_0 then there must be a non-empty ϵ-neighborhood of x_0 where the specifications fail. This assures that if we test inputs along a fine-enough subdivision of a domain we can assure ourselves of proper functioning throughout that domain there will be no surprises.

The corresponding continuous version is really the Monodromy principle, which we have discussed previously in the context of conformal mapping and analytic functions [6]. That is, to show that you have chosen the correspondingly right branches of two equivalent analytic functions, you need not check at every point, but only isolated points, one in each region bounded by branch cuts. If equivalence holds in each region, it holds throughout the complex plane, and you need not carry any restriction on domain along with the resulting expression.

3.5 What about limits?

f has a limit L as x approaches c if for every $\epsilon > 0$ there is a $\delta > 0$ **and a precision** p such that $|x - c| < \delta$ implies $|f(x) - L| < \epsilon$. We have a difficulty in that L is perhaps not rational, but might be, for example $e = 2.7182\cdots$ in which case both $f(x)$ and L must be evaluated for increased precision p.

4 Equality, or near equality

The definition of $s(x)$ works with the assumption that it is acceptable for small enough $|x|$ that we return x for $\sin(x)$. Can we say instead something about "close enough" in a definite sense? We see no way of doing this in an algorithm-independent way; it depends on the truncation error of the algorithm. Here's an analysis for $s(x)$. Since the Taylor series (below)

$$\sin x = x - \frac{x^3}{6} + \cdots$$

alternates, this suggests that if $x^3/6$ is negligible compared to x then $\sin(x) = x$ computationally. That is, we can't distinguish x from the fully-correct value. Can we quantify this? If $x - x^3/6 = x$ in precision p, or values of x where $(x^3/6)/x) < 2^{-p}$, the error is p-negligible.

So we can in fact quantify the difference between x and $\sin(x)$ as p-negligible, when $|x| < \sqrt{6}\,2^{\frac{-p}{2}}$

For $p = 23$ (single-float) $|x| < 8.457 \cdot 10^{-4}$. For $p = 53$, $|x| < 2.58 \cdot 10^{-8}$.

5 What about roundoff?

If we approach the rational computation aspect of continuity from the traditional numerical analysis perspective, we are essentially talking about truncation error of arithmetical processes, since there is no roundoff. But the floating-point implementations (unless done in unusual ways) requires consideration of the effects of roundoff. If we are in fact running a program to sum a Taylor series, certainly it matters how we compute each of the terms and how we add them up. Should we, for example, sort the terms in size order, adding up the small terms first? Should we re-order the operations (evaluating a polynomial) say using Horner's rule?

We are not contending that one step in increasing precision will produce a more accurate answer in every case. Indeed there are examples in which modestly increasing the precision provides the (same) wrong answer [8], and even where low-precision calculations accidentally hit upon the right answer. Yet ultimately, if one increases the precision, the correct answer will emerge.

6 A bottom-up approach to continuity

If we restrict ourselves to composition of addition, multiplication, power (positive), finite loops, [no overflow] we will map fp p-continuous domains to p'-continuous domains. Setting p' to an easily computable function of the exponent sizes of the inputs and p allows us to compute *exact* results Division, even if restricted to division by non-zero, is problematic unless we find some representation for repeated patterns. (E.g. in binary, 1/10 is 0.000110011001100... where the last four bits repeat indefinitely). A very interesting solution to such problems, including another view of achieving a model of continuity, is available by computing with continued fractions. This collection of techniques [12] has unfortunately acquired a reputation for inefficiency.

7 Conclusions

Arbitrary-precision floating-point computation provides a non-traditional view of the numerical analysis of mathematical function evaluation. In a model which allows ever-higher precision in calculation, we are still in fact dealing with rational numbers, we can either deal with them as rationals which can approximate a real to any desired relative error, or carry through reasoning about p-precision computations where p can be increased when needed.

References

[1] W. J. Cody, Jr. and W. Waite. *Software Manual for the Elementary Functions.* Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[2] Abbas Edalat and Reinhold Heckmann, "Computing with real numbers: (i) LFT approach to real computation, (ii) Domain-theoretic model of computational geometry in Gilles Barthe, Peter Dybjer, Luis Pinto, and Joao Saraiva, editors, LNCS, Springer, 2002. http://theory.doc.ic.ac.uk/ãe/papers/APPSEM.ps.gz also R. Heckman, "How many argument digits are needed to produce n result digits?" Electronic Notes in Theoretical Computer Science vol 24 (2000) Elsevier 21 pages.

[3] R. Fateman. "High-level proofs of mathematical programs using automatic differentiation, simplification, and some common sense" in *Proc. Int'l Symp. on Symbolic and Algebraic Computation*(ISSAC 2003), August 3–6, 2003, Philadelphia, PA" 88–94.

[4] N. Hur and J.H. Davenport, "An exact real algebraic arithmetic with equality determination, in *Proc. Int'l Symp. on Symbolic and Algebraic Computation*(ISSAC- 2000) St. Andrews, Scotland 169–174.

[5] R.B. Jones. Computing with Reals: http://www.rbjones.com/rbjpub/cs/cs001.htm

[6] "Advances and Trends in the Design of Algebraic Manipulation Systems," *Proc. Int'l Symp. on Symbolic and Algebraic Computation* (ACM/ Addison-Wesley), (ISSAC-90) invited paper, Tokyo, August, 20-24, 1990, 60–67

[7] J. Harrison. floating point trigonometric functions." *Proc. 3rd Int'l Conf. on Formal Methods in Computer-Aided Design,* FMCAD 2000. Springer LNCS 1954, pp. 217-233, 2000.

[8] S. M. Rump. In *Reliability in Computing* (R. E. Moore, ed.), pp. 109-126, Academic Press (1988).

[9] Dick Hamlet, Continuity in software systems, *Proc. ISSTA 2002 International Symposium on Software Testing and Analysi* July 22-24, 2002, Roma, Italy, 196–200.

[10] P. Potts. "Efficient on-line computation of real functions using exact floating point," http://www.purplefinder.com/p̃otts/efficient.pdf (1998)

[11] J.R. Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Predicates for Computational Geometry," *Discrete & Computational Geometry 18*:305-363, 1997. http://www-2.cs.cmu.edu/ quake/robust.html

[12] J. Vuillemin, "Exact real arithmetic with continued fractions," *IEEE Trans. on Computers 39*, No 8 Aug. 1990 1087–1105.

9/30/03 RJF