High-level Proofs of Mathematical Programs Using Automatic Differentiation, Simplification, and some Common Sense

Richard Fateman
EECS Department
University of California, Berkeley

ABSTRACT

One problem in applying elementary methods to prove correctness of interesting scientific programs is the large discrepancy in level of discourse between low-level proof methods and the logic of scientific calculation, especially that used in a complex numerical program. The justification of an *algorithm* typically relies on algebra or analysis, but the correctness of the *program* requires that the arithmetic expressions are written correctly and that iterations converge to correct values in spite of truncation of infinite processes or series and the commission of numerical roundoff errors. We hope to help bridge this gap by showing how we can, in some cases, state a high-level requirement and by using a computer algebra system (CAS) demonstrate that a program satisfies that requirement. A CAS can contribute *program manipulation, partial evaluation, simplification or other algorithmic methods*. A novelty here is that we add to the usual list of techniques *algorithm differentiation*, a method already widely used in different contexts (usually optimization), to those used already for program proofs. We sketch a proof of a numerical program to compute sine, and display a related approach to a version of a Bessel function algorithm for $J_0(x)$ based on a recurrence.

Categories and Subject Descriptors

I.1.1 [**Expressions and Their Representation**]: Simplification of Expressions; I.2.2 [**Automatic Programming**]: Program verificationI.1.1Expressions and Their RepresentationSimplification of Expressions

Keywords

Bessel, sine, recurrence, proof, differentiation

General Terms

Theory, Languages, Design, Algorithms

1. INTRODUCTION AND MOTIVATION

If you are given a functional program, how can you determine if it does what it is supposed to do? This is a complicated question that requires a specification of "what it is supposed to do." Presumably this specification is separate from the program itself to avoid the tautology "It does what it does." Answering such a question seems to require a kind of proof that the specification is equivalent to the program. Often it is easier to prove subsidiary properties such as "free of data access errors."

For scientific programs, sometimes claims for correctness are not much help in that they make critical assumptions (e.g. "assuming no overflow happens") exactly when you would like to prove that assertion (e.g. overflow cannot happen).

Another problem discussed at length in software engineering is that specifications are prone to have bugs too: and while one can test and debug programs, how is one to check the specification? One solution is to derive the program from a specification: it may force them to agree, but that does not necessarily have any bearing on some "user requirements" which are perhaps only informally stated, and may differ from the specification when examined closely. There is a substantial literature on software engineering; almost none of it has bearing on scientific computing [14].

In our own explorations about proof of numerical or symbolic programs, we have concentrated on constructive methods, rather than existence proofs. We have looked for tools that work on programs themselves and prove properties that can be very simply stated. An early example of such an approach is recursion induction [8] by which McCarthy demonstrated a simple proof, based on the text of the Lisp `reverse` program, that for any list `s` it is the case that `reverse[ reverse[s] ] =s`. Of course a correct `reverse` program must have additional properties.

To make our job of reasoning about programs more compact, we will occasionally use, without explicit formal justification, some properties of the ordinary algebra of numbers. We mention this in our introduction to alert the reader that there is a danger here: ordinary computer hardware arithmetic domains do not follow such rules (e.g. "$1 + x = x$ implies $x = 0$" is false for floating-point numbers 1.0 and x when x is small but *not* zero.) We must be cautious that our *specific* proofs work not only with (exact) integers and rationals, but when necessary with approximate number models

as may be used in the computers running the programs in question. Using an axiomatization of IEEE floats is possible but requires substantial careful work [7].

We will write and informally prove software that does not use hardware floating-point numbers: Arithmetically correct software for *rational* numbers is easily available in a CAS or any of several functional-style languages or ANSI Standard Common Lisp; other languages can use arbitrary-precision libraries (GMP, MPFUN) to provide arbitrary-precision integer arithmetic, often extended to "big" floats and rational numbers.

we will in general settle for informal proofs that –using exact arithmetic– our programs produce the correct answers, and use general arguments that we might as well use floats. To be rigorous, a floating-point proof would justify every arithmetic operation. Why bother with non-rigorous proofs? Should we even use the word *proof*? Two reasons: the use of such methods may identify erroneous programs nevertheless, and even an endlessly tedious proofs is always "relative" to some technology. We can only prove C program correct relative to the assumption of a C compiler, which assumes correctness of hardware, and is even then relative to the absence of cosmic rays affecting run-time behavior.

We hope, however, that the techniques in this paper will add to automating proofs while also suggesting locations of the remaining shortfalls.

The point of this paper is to show *some techniques for reasoning about programs*: The two examples are a program that computes $\sin(x)$, and a program that computes a Bessel function $J_0(x)$. The first (sin) was inspired by seeing the marvelously small program in HAKMEM [3] and wondering in what ways it can be shown correct. The second (Bessel) was a result of an initially fruitless attempt to use the same technique on this more complicated computation, and a fallback to a simpler approach.

Our hope is that the reader will be inclined toward the use of similar techniques to produce partial proofs more widely. Among the techniques here, we use differentiation of an algorithm [2] for purposes of proof, which we think is novel[1]. There continue to be substantial efforts in automated reasoning described in a growing literature[2]. We feel the approach in this paper of incrementally hand-crafting some knowledge of mathematics through simplification provides (at a minimum) a significant additional technology to these other paths in automated reasoning and for some cases a plausible alternative.

2. PREVIEW OF THE APPROACH

Consider a program p which is alleged to compute a mathematical function f with several properties. Manipulate p by partial execution or symbolic simplification to confirm that those same properties hold over some appropriate domain. If the properties are sufficient to *define* f, as say, p is the unique solution to f's defining differential equation, or solves a defining recurrence, then we have confirmation that the program computes that mathematical function. We refined our thoughts along these lines by considering a particular program described in the next section.

3. A PROGRAM TO PROVE

Here is a surprisingly small program which, for suitable ϵ and x, and suitable arithmetic operations, computes an approximation to $\sin(x)$. It is given in Mathematica syntax, just to be concrete. It could be written in Lisp or C or Java. It can be written in Fortran only if the version of Fortran allows recursion.

```
s[x_] := If [Abs[x] < eps,
             x,
             Block[{g = s[-x/3]}, 4*g^3 - 3*g]]
```

Some facts about the program can be deduced by testing and reasoning. For example, if `eps` is 0.00000006 (6.0d-8) or less then the agreement between `Sin[x]` and `s[x]` is good to double-precision (16 decimal places) as long as `x` is in $[-\pi/8,\ \pi/8]$. It is even good to 14 places in $[-100, 100]$. For single precision, `eps` can be as large as 0.002 and get full-accuracy answers in this range.

Your reaction to the one-line program is likely to also be mild surprise: it looks far too small to compute anything as complicated as $\sin(x)$. It becomes more plausible when we remind you of this multiple-angle formula:

$$\sin x = 4\sin^3(-x/3) - 3\sin(-x/3)$$

There are similar recursive formulas for other elementary functions and correspondingly similar programs for them ([3], items 158–160). Justifications for most of them could follow the same template we have set out here.

A reasonable simplification to this program is to assume that it will be used when its argument x is in some small interval around the origin. This allows us to reason better about overflow and accuracy. A separate preliminary range-reduction program can assure this. Proving this correct would require a separate argument, as for example in Harrison [7].

Oddly enough, the same kind of program `s`, and often the exact same program text, appropriately generalizing the termination criterion, can be used for interval computations, complex numbers, rational arithmetic.

Our primary question is: "What computation can we make to confirm that the program is correct?" To a lesser extent we may ask "Can we get a computer to reprod uce the proof?" (We must also be fairly specific about what it means to be correct.)

[1]The novelty is not "algorithm differentiation," or AD, *per se* which has quite a large literature, but using AD as a proof technique. For a reader unfamiliar with AD, we recommend setting aside this paper and looking at the cited reference.

[2]Here are three recent citations where an interested reader can follow references back through further links. B. Shults' dissertation [13], W-T Wu's monograph [15], or A. Bundy's survey [4].

We have programs that can mechanically convert the program (at the moment we prefer to begin with an equivalent Lisp form) to a version that computes its first and second derivatives too. This is reasonably well understood as evidenced by the extensive work on automatic differentiation of programs. Some 18 programs are mentioned in an on-line collection [2], not even including the one we developed for differentiation of a Lisp program.

For reference, here is the Lisp version of the original `s`. As a concession to those who are more comfortable with infix syntax, we will continue with the Mathematica displays.

```
(defun s(x)(if (< (abs x) eps)
               x
               (let ((g (s (/ x -3 ))))
                      (+ (* 4 g g g) (* -3 g)))))
```

One kind of algorithm differentiation program is illustrated below where we convert a function of one argument $s(u)$ into another function which computes the value of $s(u)$ as well as its first and second derivative with respect to some (implicit variable) say x.. This new program we call `ts` takes 3 arguments, u, and the first two derivatives of u with respect to x which may initially be equal to u. We can differentiate the Lisp program using an AD tool, the details of this technology constitutes a distraction at this point. Presumably it could be done automatically on a Mathematica or Maple [12] program too. Here is what it would probably look like in Mathematica:

```
(* compute s(u), s'(u), s''(u)  given u, u', u'' *)
ts[u_,up_,upp_]:=
    If [Abs[u]<eps,{u,up,upp},
       (* return a triple of s, s', s'' *)
          Block[{r=ts[-u/3,-up/3, -upp/3],g,gp,gpp},
             g=r[[1]];gp=r[[2]]; gpp=r[[3]];
       (* return a triple of three items *)
          {4*g^3-3*g,                (*the formula *)
          12*g^2*gp-3*gp,             (* s' *)
          24*g*gp^2+12g^2*gpp -3*gpp} (* s'' *)]]
```

or in Lisp

```
(defun ts(u up upp)
 (if (< (abs u) eps) (values u up upp)
  (multiple-value-bind (g gp gpp)
    (ts (/ u -3) (/ up -3) (/ upp -3))
    (values (+ (* 4 g g g) (* -3 g))
            (+ (* 12 g g gp)(* -3 gp))
            (+ (* 24 g gp gp)
               (* 12 g g gpp)
               (* -3 gpp))))))
```

Program note: `multiple-value-bind` in Common Lisp takes the three values returned by the `values` construction in the recursive call to `ts`, and binds them to `g`, `gp` and `gpp`, without the use of the temporary `r` and the extraction of parts needed in the Mathematica program.

Observe that the operations in this function are rational operations and can be done exactly on rational inputs. We claim the formula for `ts[x,1,0]` computes a triple: $\{\sin x,\ \cos x,\ -\sin x\}$. We can show that this recursive function call returns a triple $a(x), b(x), c(x)$ where $a(x) = -c(x)$, and $c(x)$ is the second derivative of $a(x)$. It therefore solves the differential equation $a'' + a = 0$. Combined with knowledge that $a(0) = 0$, we have convincing evidence we are computing $\sin(x)$.

This demonstration requires some work. We differ from researchers who are using purely logical transformations (using any of the numerous logical frameworks dating back to at least 1968 with Automath [5] continuing to the present COQ [6]) in that we emphasize building a system in which proofs are based on *simplification* of functions. This does not preclude an approach using multiple tools (including lessons learned from various logical frameworks) in a total environment.

The actual proof of this result has four parts indicated below and fleshed out in a subsequent section:

1. `ts` is a valid function of its arguments: never overflows or underflows.
2. `ts[0,1,0]={0,1,0}` confirming that $\sin 0 = 0$, $\cos 0 = 1$ and $-\sin 0 = 0$.
3. The first and last results from `ts` have the same absolute value but differ by a sign. It is a solution of the harmonic equation with appropriate initial conditions for $\sin x$.
4. Proof of termination (easy if we can determine the conditions: Given non-zero ϵ, as n increases, eventually $|x/3^n| < \epsilon$.). If we use this termination condition we must also present an argument that $|x| < \epsilon$ means $\sin x = x$, good enough for us. That is, (if we are using an inductive argument) the base case is not just that for small x we return x but that this is also $\sin x$.

4. SELECTED DETAILS OF THE PROOF

4.1 Valid, bounded

It is possible to discuss programs to evaluate polynomials at great length [11].

Evaluated exactly at rational numbers in a CAS, no over- or underflow is possible. A version of continuity in the "epsilon-delta" sense, but using rational numbers is supported. For floating-point computation, discrete steps prevent us from finding an appropriate $\delta > 0$ for every $\epsilon > 0$. We could however argue that the existence of a derivative at every point (at least at every denumerable floating-point number we can test) suggests an alternative notion of continuity "for all practical purposes".

We have to look at the particular polynomials being evaluated to see if they are bounded or continuous "enough" in the appropriate range.

Floating-point truncation in dividing by 3, cannot overflow. (And for exact rational arguments we do not even need to

make this excuse.) A worst-case interval analysis for each of the polynomials such as $4*g^3 - 3*g$ with $|g| < 1$ shows they don't overflow in ordinary float arithmetic.

4.2 Initial conditions

By simplification or execution, we can see that the function at $x = 0$ has the required derivatives since we are calling `ts[x,1,0]`.

4.3 Confirmation of derivative

Here we are really looking for a proof that `s[x]` is a solution to $y'' - y = 0$ by examining the second derivative of `y=s[x]` as computed by `ts`. Make the **recursion induction assumption**: if x is small enough the `ts` program returns the triple of $\{g = \sin(-x/3),\ g' = \cos(-x/3)/(-3),\ g'' = -\sin(-x/3)/9\}$, namely the value of $\sin(u)$, and two derivatives with respect to x of $\sin(u)$ where $u = -x/3$.

Then we need to know that the three expressions in the body of `ts`

$$\{4*g^3 - 3*g, 12*g^2*g' - 3*g', 24*g*g'^2 + 12g^2*g'' - 3*g''\}$$

are mathematically valid and within our notion of "computationally close enough," $\{\sin(x),\ \cos(x),\ -\sin(x)\}$.

In Mathematica (similar results can be derived in other computer algebra systems), a simplification to the triple pligged into the 3 forms does this:

```
m[g_, gp_, gpp_] := {4*g^3 - 3*g,
                     12*g^2*gp - 3*gp,
                     24*g*gp^2 + 12g^2*gpp - 3*gpp}

FullSimplify[m[Sin[-x/3], Cos[-x/3]/(-3),
               -Sin[-x/3]/9]]
```

This last command returns `ans={Sin[x], Cos[x], -Sin[x]}`, or more particularly if y =`ans[[1]]` and y''=`ans[[3]]` then `ans[[1]]+ans[[3]]` is the computation of the LHS of the differential equation which will simplify to zero, confirming that our program computes a solution to $y'' - y = 0$.

How much of this proof can be done by a computer?[3]

Proving item 3 by computer: all we need is a simplifier to multiply out various forms and deftly apply identities such $\sin^2 u + \cos^2 u = 1$. Fortunately `FullSimplify` is adept at this. In this case we relied on the existing collection of transformations; in other cases we must anticipate having to carefully add to the collection. In so doing we are providing knowledge (dare we call them "theorems"?) in the form of rules or algorithmic reductions. This knowledge, properly encoded, would be re-usable in future proofs or simplifications.

- Since the argument x is representable, then so is $x/3$: exactly if rational arithmetic is done, approximately for floats where it is subject to truncation and to (gradual) underflow. (We considered other formulas using floats and $x/2$ where we could make a stronger statement that $x/2$ is exactly representable subject only to (gradual) underflow in IEEE binary arithmetic. This complicates the program since we need sin and cos for half-angle formulas.)
- If `ts` returns 3 numbers that correspond to {sin, cos, -sin}(x), then each is a representable numbers in [-1.0, 1.0].

Any of the number systems we are considering can compute any of the formulas without overflow since the largest possible sub-expression is by hypothesis an accurate approximation to a function of $-x/3$, sin or cos; the correct values are bounded within $[-1, 1]$. If we use the `s` program only for small real x (We need compute only for x values in $[-\pi/8,\ \pi/8]$, by argument reduction and reflection since $\sin(x)$ is odd and periodic; in that interval it is bounded in absolute value by about $[-0.3827, 0.3827]$).

Argument reduction does not work for complex numbers. For a pure imaginary input, `s[z*I]` this program computes $i\ \sinh(z)$ which is not periodic. For large z this grows as $\exp(z)/2$ and can certainly overflow. Any proof must account for this prospect, and an automated proof of the bound must make use of the fact that the argument x is small.

4.4 Termination

There are several ways of terminating this recursion. We can pre-compute some $\epsilon(p)$ for which further computation (in precision p) is pointless. In this specific example, we could cease when $x - x^3/6$ is indistinguishable from x [3]. Since each recursion level reduces the input by a factor of 3, the number of recursions is $n = -\log_3(\epsilon/x)$ which is clearly finite as long as $\epsilon > 0$. Here's a Lisp program to compute ϵ.

```
(defun eps(p)
  ;; Return a value eps such that 1-x^2/6 = 1
  ;; in the same precision as p
  (assert (floatp p))
  (setf p (if (= p 0) (1+ p) (/ p p)))  ; 1.0 or 1.0d0
  (do ((x p (/ x 2)))
      ((= 1 (- 1 (/ (* x x) 6))) x)))
```

This works for any finite-precision arithmetic, but returns an error for an attempt to use rational arithmetic.

5. OTHER APPLICATIONS

Similar recursive remarkably short formulas for sinh, tan, tanh, etc. as well as exponential and log (neatly making use of square-root) are described in HAKMEM. [3]. We will not, therefore, examine in detail these elementary functions.

Additional grist for this mill includes any functions with argument-reduction relations. These do not seem so likely among the special functions, so if this kind of technique is to work, other approximate recursive convergent program generation ideas may be needed. In the past we have explored "bad" numerical techniques that are nevertheless surprisingly useful symbolically, including Picard iteration or algebraic Newton iteration (p-adic) [9].

[3]Assuming a computer has no hands to wave.

Before proceeding down this path, let us first observe that CAS are quite good at generating Taylor series, which can be converted to programs and related rather directly to the defining differential equations.

5.1 Bessel Functions from Taylor Series

For example a CAS can automatically write a program from the 14-term truncated Taylor series around zero for the Bessel function of the first kind $J_0(x)$:

```
b[x_]:=1-x^2/4+x^4/64-x^6/2304+x^8/147456
 -x^10/14745600+x^12/2123366400-x^14/416179814400
```

We can also automatically produce the first and second derivatives of this expression (or trivially, here just the body of the program). If we substitute into the usual definition:

$$x^2 b'' + x b' + x^2 b = 0 \tag{1}$$

we get a result of $-x^{16}/416179814400$ which suggests that if $-2.4^{-12} * x^{16}$, is negligible compared to $b(x)$ then the calculation (modulo roundoff) is valid. Our demonstration is not adequate to show that a better expression for `b(x)` is the Horner's rule version:

```
b[x_]:=(x^2*(x^2*(x^2*(x^2*(x^2*(x^2*(196-x^2)
   -28224) +2822400) -180633600) +6502809600)
   -104044953600) +416179814400) /416179814400}
```

5.2 Bessel Functions from Recurrence

A more realistic approach to Bessel (or other special functions) is to use a variety of numerical schemes. For example, this often includes matching asymptotic forms over different sections of the domain, using approximating polynomials, or some version of a traditional reduction recurrence (see for example Abramowitz/Stegun [1]). For $J[n](x)$ one scheme is to use a downward recurrence that runs not on the argument, but the index of the Bessel function.

$$J_n(x) = 2 * (n+1)/x J_{n+1}(x) - J_{n+2}(x) \tag{2}$$

For $n >> x$ we can approximate $J_n(x)$ by zero, and somewhat perversely, $J[n-1](x)$ by 1 or perhaps $1/k$ for some k. The arbitrariness of this (what is k?) disappears after a "normalization" in which k is computed.

$$1 = J_0(x) + 2 * (J_2(x) + J_4(x) + \cdots). \tag{3}$$

The definition of the Bessel program for J_0 for $x > 0$ can be written in Lisp as some variant of the following brief text. (variants are mostly in computing the termination condition or in making the arithmetic "generic".)

```
(defun besj0(x)
      ;; J[n](x) = 2*(n+1)/x *J[n+1](x) -J[n+2](x).
      ;; normalize by R= 1= J[0](x)+2*J[2](x)+.....
  (let ((1/x (/ 1 x)) (x10  (+ (* 3/2 (abs x)) 15)))
       ;; x10 is one stopping limit when J[v](x)=0
  (labels
   ((m (n)
      (if (> n x10 ) ;; makeshift termination..
          (values 1 0 0)
          ;; return j[n](x), j[n+1](x),  normalized,
          ;; and partial R sum
        (multiple-value-bind (jnp1 jnp2 sum)
            (m (1+ n))  ;; get jn+1, jn+2, sum so far
          (let ((jn (- (* 2 (1+ n) jnp1 1/x ) jnp2)))
 ;; J[n](x) = 2*(n+1)/x* J[n+1](x) -J[n+2](x)
            (values jn
                    jnp1
                    (if (evenp n) ;;add J_n if n is even
                      (+ sum jn jn)
                      sum)))))))
 (multiple-value-bind (a b c) (m 0) (/ a (- c a))))))
```

Some observations: This program, *without change*, will use exact rational arithmetic if the input is an integer or rational number, in which case its accuracy is determined by the termination test and also the proximity of the argument to a zero of J_0. In such an environment, overflow which could otherwise be a problem in accumulation of the normalization sum for large argument, is impossible. (Arithmetic on rational numbers with large numbers of digits in numerator and denominator can be slow, of course.) The program can also be used, *without change*, for complex arguments. With appropriate definition of the arithmetic, it can be used with "bigfloat" packages, interval arithmetic, etc.

The numerical accuracy of this program depends on the terminating condition being set right, as well as the precision of the arithmetic. This cannot be done infallibly *a priori* for fixed relative precision (e.g. one unit in the last place) based on loose estimates. Arguments near a zero of J_0 will need far more iterations for the same relative accuracy, as well as more careful arithmetic. For example, if we choose 93170664/38743211 = 2.404825557695773..., as a good approximation to a zero of J_0 we can evaluate this function by running 19 iterations to give a rational-number approximation with a huge numerator (145 digits) and huge denominator (161 digits). Converting this to double-precision gives -2.4350447538776332d-17 which is accurate to all places. Starting with the float version, 12 iterations gives 0.0; continuing to more iterations gives numbers like 14: `-4.8951805787910495d-17`.

That is, in this case roundoff error in the arithmetic eventually dominates the computation, but disaster is avoidable by using higher precision or exact rational arithmetic.

A technique sometimes used for estimating the number of terms needed is based on reversing the recurrence and running the (unstable) forward recurrence. If this returns to the two inputs, then the arithmetic is not problematical.

This termination condition does not enter into our proof: it would be an additional computation to see if our choice of number of iterations is high enough and our working precision is high enough (or using infinite precision rationals). In our proof we use as an inductive basis that we know some value of $J_n(x)$ "well enough" to say that it is zero relative to $J_{n-1}(x)$. It is also much more convenient to analyze the rational arithmetic version.

Proving the correctness of this program under the circum-

stances above can be based on some set of Bessel function identities equivalent to the definition, including twice differentiating this algorithm and substituting into the defining equation. Bessel himself seemed to view the definition of J_0 as an integral, but given the large collection of Bessel "facts" alternative mathematical definition abound. For example, if one is willing to just allow that Bessel functions are equivalently defined by this recurrence, plus satisfy the normalization, we are quite close to a proof already, modulo the (significant) numerical approximation arguments.

Another proof approach which we pursued was to demonstrate that the result of the recursion on a symbolic result ($k+1$ times iterated) is identical up to $O(k)$ with the Taylor series. See Appendix I.

Unfortunately the Bessel function needs to be approximated over a wide domain (not just one period), so a more principled numeric approach may need to be subject to proof. In this case we suggest that the computation take an extra argument or two: the nature (absolute or relative) and the size of the allowable error in the answer, and optionally a guess on how many iterations to take. With a little more analysis we could take a better guess based on the argument and relative error. The number of additional terms is proportional to $\sqrt{z}$ for z bits of accuracy. Computing the proper working precision would depend on proximity to a zero of J_0.

Programs along these lines (but clearly based on the one above):

```
(defun besj0(x count)    ;use ‘‘count’’ terms
  (let ((1/x (/ 1 x)))
 (labels
   ((m (n)
     (if (> n count)(values 1 0 0);; simple term.
       (multiple-value-bind
            (jnp1 jnp2 sum)
              (m (1+ n))
         ;; if sum is getting too big,
         ;;  make everything smaller
         (when (and (floatp x)(> (abs sum) 1.0d10))
           (setf sum (* sum 1.0d-10)
                 jnp1 (* jnp1 1.0d-10)
                 jnp2 (* jnp2 1.0d-10)))
          (let ((jn (- (* 2 (1+ n) jnp1 1/x ) jnp2)))
            (values jn jnp1
               (if (evenp n) (+ sum jn jn) sum)))))))
(multiple-value-bind (a b c)(m 0)
                       (/ a (- c a))))))

;; possible usage for relative error
(defun superbesrel(x relerror
                     &optional
               (guess (round (+ (* 3/2 (abs x)) 10))))
  (let* ((first (besj0 x guess))
         (second (besj0 x (+ 2 guess)))
         (diff (* 2 (abs (- first second)))))
    (if (< relerror (abs(/ diff second)))
        (superbesrel x relerror (* guess 2))
      second)))
```

```
;; alternative for absolute error
(defun superbesabs(x abserror
                     &optional
               (guess (round (+ (* 3/2 (abs x)) 10))))
  (let* ((first (besj0 x guess))
         (second (besj0 x (+ 2 guess)))
         (diff (* 2 (abs (- first second)))))
    (if (< abserror (abs diff))
        (superbesabs x abserror (* guess 2))
      second)))
```

In either case we can consider any input as though it were an exact rational input and view the result as acceptable only if it passes a test for an absolute or relative error bound. Testing that two successively more refined answers differ by less than half that bound is suggestive but not proved in the face of floating-point roundoff. (Even in the rational arithmetic case it also requires some demonstration, omitted here, that carrying out an additional 2 terms in the iteration provides essentially two terms more in an alternating Taylor series.)

By providing the accuracy requirement as an additional input to the routine, we can produce a more general routine than is possible with the traditional view (in which the types of the input govern the accuracy: double-precision input leads to an answer that is almost correct to double-precision). After all, we might wish to have single-precision accuracy for a double-precision argument, or a rational argument, etc.

We have not demonstrated the absence of overflow: indeed, trying to find too high a precision result while using low-precision arithmetic can lead to overflow in the `sum` computed for normalization. Counteracting this may be achieved by the modification inserted above with `superbes`. Another possible trick would be (for example) to specialize a version of `besj0` for double-float that returns a base case for J_{n-1} of a very small number (conventionally given the name `least-positive-double-float` in Common Lisp) instead of 1, and similarly for `single-float`. That way the whole exponent range is used before renormalization.

The possible extension of this technique: of provable (possibly recursive) high-precision evaluation (possibly using unconventional rational arithmetic) may provide a needed relief from a situation in which carefully crafted programs have to be devised to get around the limitations of conventional floating-point hardware.

Although not made of the same kind of recursive argument-reduction as the sin program, we hope it may still be worthwhile to consider automating validation of some computations; the programs from symbolically generated Taylor series are possible but current CAS tools do not provide information on convergence. We discuss some automatic generation techniques that may assist in generating correct software using memoized sequences or software "pipes" in another paper [10].

6. CONCLUSION

As computers have become faster, the cost of special less-efficient arithmetic (including for example interval or rational arithmetic) may fade in significance when the trade-off is to have an assurance of validity, combined with relatively quick generation of specially-built procedures on different argument types. Note, for example, the complex-valued version of the sine program is identical to that given above. $\sin(ix)$ computes $i \sinh x$. Even the interval-valued version is essentially the same program statement, although the termination test then must be restated for intervals. We don't expect that such programs will be used indefinitely in public libraries: in the marketplace of programs correct programs may be driven out by faster hand-coded programs. (Sometimes they are driven out by free, less-robust code as for example published in some "recipe" books). For one-time generation or occasional use, the correctness arguments might prevail.

Although we have only sketched the nature of proofs of these examples, we hope that we have provided some suggestive material on the relatively high level of computation and the general form of proof we are looking for. We see a need to supply a modicum of rigor suitable for program validation for these tests. There is clearly more work to be done, even on these examples in characterization of algorithms using floating point numbers, as well as nailing down the nature of precision.

7. ACKNOWLEDGMENTS

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks also to the ISSAC referees for providing the leeway for extensive revision of an initially hasty draft.

8. REFERENCES

[1] M. Abramowitz, Stegun, I. (eds.)*Handbook of Mathematical Functions*, Dover Publications, 9th edition, 1972.

[2] A Collection of Automatic Differentiation Tools. `http://www-unix.mcs.anl.gov/autodiff/AD_Tools/`

[3] M. Beeler, Gosper, R.W., and Schroeppel, R. HAKMEM. MIT AI Memo 239, Feb. 29, 1972. Retyped and converted to html by Henry Baker, April, 1995.

`http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item158`

[4] A. Bundy. "A Survey of Automated Deduction". Lecture Notes in Computer Science volume 1600, Springer-Verlag, 1999.

[5] N. de Bruijn. A survey of the project AUTOMATH, in J. Seldin and J. Hindley, eds, 'To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism', Academic Press, 1980,pp. 579—606.

[6] G. Dowek, Felty A., Herbelin H., Huet G., Murthy C., Parent C., Paulin-Mohring C. and Werner B. [1993], "The Coq proof assistant user's guide," Rapport Techniques 154, INRIA, Rocquencourt, France. Version 5.8.

[7] J. Harrison. "Formal verification of 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] J. McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Comm. ACM 3* no 4 (1960) p 184–195 `http://www-formal.stanford.edu/jmc/recursive/recursive.html`

[9] Richard Fateman."Series Solution of Algebraic and Differential Equations: a Comparison of Linear and Quadratic Algebraic Convergence," Proc ISSAC-89, ACM Press.

[10] R. Fateman."Compiling functional pipe/stream abstractions into conventional programs: Software Pipelines," on-line notes.

[11] R. Fateman."Programs for evaluating polynomials," on-line notes.

[12] M. B. Monagan and W. M. Neuenschwander, "GRADIENT: algorithmic differentiation in Maple," Proc. ISSAC-93, ACM Press. p. 68–76.

[13] B. Shults. "Discoveries and Experiments in the Automation of Mathematical Reasoning" PhD dissertation, Univ. Texas (Austin) Dec. 2002 http://www.cs.wcu.edu/ shults/IPR/diss-new.pdf

[14] I. Sommerville. *Software Engineering* (6th edition), Addison Wesley, 2000.

[15] Wu Wen-tsun. *Mechanical Geometry Theorem-Proving, Mechanical Geometry Problem-Solving and Polynomial Equations-Solving* (Kluwer, 2001).

9. APPENDIX I

A demonstration that the recursive program for J[0](x) run to 14 terms is an approximation to 13 terms of the same function as the series for the Bessel function (in Mathematica).

```
{Clear[J,m,x];
m[n_]:= Simplify[2*(n+1)/x*m[n+1] -m[n+2]] ;
m[14]:=0;  m[13]:=1; (*arbitrary *)
R=m[0]+2*(m[2]+m[4]+m[6]+m[8]+m[10]+m[12]);
(*J[0](x) should be m[0]/R *);
Print["J[0](x) as computed with 14 levels of recursion: ",
       Simplify[m[0]/R]];
Print["Subtract this answer from Bessel Series: ",
       BesselJ[0,x]-Series[m[0]/R,{x,0,14}]]}
```

J[0](x) as computed with 14 levels of recursion:

$$3643696742400 - 840853094400\,x^2 + 40144896000\,x^4 - 663552000\,x^6 + 4354560\,x^8 - 10752\,x^{10} + 7\,x^{12} \;/\; 3643696742400 + 70071091200\,x^2 + 34560\,x^8 + 192\,x^{10} + x^{12}$$

Subtract this answer from Bessel Series:

$$\frac{-x^{14}}{714164561510400} + O(x)^{15}$$