DRAFT!! Comments on Unrestricted Algorithms for Bessel Functions in Computer Algebra: Arbitrary Precision, The Backwards Recurrence, Taylor Series, Hermite Interpolation

Richard J. Fateman
University of California
Berkeley, CA 947220-1776

June 4, 2016

Abstract

We explore various ways of implementing "unrestricted algorithms" [3] for approximating Bessel (J) functions. An unrestricted algorithm for a function $f(x)$ provides a result to any requested precision in the answer. The emphasis is on higher-than-normal precision with the precision specified as an extra argument to the function. That is, the precision is specified at run-time. We require that the algorithm provide at least the requested number of correct digits, contrary to some existing codes which provide only "absolute error" near critical points. We use J_0 of real non-negative argument as an example, although much of the reasoning generalizes to other Bessel functions or related functions.

Since it is plausible that there will be requests for additional values of J_0 at the same (high) precision at a collection of nearby arguments, we consider implementations that cache certain re-usable key constants (namely zeros of J_0 near the argument values).

1 Introduction

The methods we look at for computing Bessel functions (The notation for Bessel functions of the first kind is $J_\nu(x)$) include: (a) The now traditional algorithm using a 3-term formula relating Bessel functions of different order using the defining recurrence in the "backwards" direction. (b) Taylor series expanded at zero or especially at other locations. (c) Two-point Hermite interpolation, a kind of generalization of Taylor series, and (d) asymptotic formulas for large argument. To qualify as an"unrestricted" algorithm, the procedure must in essence be a function not only of the obvious arguments ν and x but also a run-time-specified precision p. This is the number of digits to which the answer is correct. Conventional fixed-degree rational function approximation is suitable only if the precision requirement is limited and can therefore be accounted for when the approximation formula is derived. Even so, such formulas tend to have large *relative* error near zeros of the function.

While we are using Bessel functions as the first "less familiar" function in relatively common use, we do not require the reader to be especially familiar with Bessel functions (or for that matter, Hermite interpolation). In particular, some of the methods used here can be applied to other scientific functions of substantial interest. See Gautschi [6] for a survey of recurrence methods, and Corless [4] for a discussion of a similar sort involving computer algebra systems, arbitrary precision, recurrence methods and Airy functions. The standard concise reference for other methods, especially asymptotic series is the Digital Library of Mathematical Functions (DLMF)[1]

[1]DLMF http://dlmf.nist.gov/10.17

A Bessel function $J_n(x)$ can be visualized as a kind of sinusoidal wave of decreasing amplitude as $|x|$ increases, asymptotically approaching zero. Also as $|x|$ increases, the spacing between the zero-crossings approaches a constant, namely π. Details and plots of J_0, J_1, as well as relations to differential equations, integrals, and other functions can be found online in the DLMF and in reference books e.g. Watson[15].

Given an environment of a computer algebra system[2] eases the burden for development of unrestricted algorithms. Support for representation and arithmetic on arbitrary-precision floating-point numbers with effectively unlimited exponent range, it is often possible to construct fairly naive unrestricted algorithms. One method is to use brute-force expansion of Taylor series using increasingly higher precisions, adding additional terms until some termination condition is reached. There are two problems with this: (a) It is displeasing aesthetically (and sometimes practically) to do what appears to be far more work than necessary to get the right answer, and Taylor series expanded at zero can present substantial work; (b) Determining a correct termination condition may be more subtle than appears at first glance. The NumGFun package (in Maple) [10] provides a general framework for Taylor series expansions of "D-finite" functions. These include Bessel functions.[3]

Among the existing Bessel function programs is the MPFR library, `http://www.mpfr.org` an open-source implementation whose algorithm is also documented and"proved"[4]. It uses a Taylor series expanded around zero, but when the value of the argument is sufficiently large (compared to the index and the required precision) it switches to an asymptotic formula. MPFR attempts to sum the terms correctly by boosting the working precision relative to the precision p expected in the result. It adds to the precision the log of the ratio between the largest term in the series and the first one plus $p + 2\log_2 p + 3$. This is a good estimate, but testing shows it doesn't always quite do the job. We tested on $z_{40} =$ 953131103962007545291/23793523728624063229. This is an approximation to the root of J_0 near 40.0058... Computing $J_0(z_{40}+1/100)$ to 300 bits yields a result -0.00126 ... with 268 bits (about 80 decimal digits) correct, so the 300 bit requirement is not met even "near" z_{40}. Moving much closer to the zero, and computing $J_0(z_{40})$ to 300 bits yields a result -2.145...E-41 which is correct to only 169 bits. Examining the coefficients in the Taylor series shows rather large terms (e.g. the 23rd term is of magnitude 10^{15}). Such terms *must mostly cancel since the first non-zero digit in the answer is 41 digits to the right of the decimal point.*

By contrast, if we consider expansion for $z_{40} + 1/100$ around the approximate zero at z_{40} rather than around zero, it takes rather few terms (about 30) and is accurate to 90 decimal places.

Guaranteeing an answer within the requested relative error (number of correct digits) is one of the criteria for this current work; it is not fulfilled by MPFR in this case, nor Mathematica as it turns out.

When we tested the numerical implementation of `BesselJ` in Mathematica 8.0, we found that the value for `N[BesselJ[0,z40],90]`, (corresponding to about 300 bits), returned

-2.14544245178072877346184333649845274003117394888568875414093990247*68...* E-41

which is good to only about 67 decimals, not 90. We indicate the first two erroneous digits in italics, and omit the additional following digits. correct bits by requesting much higher precision or by specifying from `N` what Mathematica calls `Accuracy`, of 90 digits plus the count (41) of zero digits to right of decimal point: `N[BesselJ[0,z40],{Infinity,131}]`.

The algorithm used by Mathematica 8.0 is not (so far as we can tell) described in the open literature.

The algoritm was apparently changed after Mathematica 8.0, because Mathematica 10.0 appears to provide a correctly rounded result, in any software-implemented floating-point: even requesting 10 digits explicitly gives -2.145442452e-41; using the machine-precision routine gives -3.37648e-17, an astounding 24 orders-of-magnitude error.

---

[2]Our illustrations are given in the free, open-source Maxima system, a descendant of the early Macsyma system.

[3]The initial emphasis here seems to be on harnessing theoretically-fast but general algorithms, using exact arithmetic when possible to find isolated numerical values to some required absolute error bound. Most likely this won't be economical unless thousands or millions of digits are required, and would not be a fast approach for evaluations of the same function at many different points. Future research could address finding good approximations. (personal communication, 9/8/2012 MM)

[4]`www.mpfr.org/algorithms.pdf`

We also tried Maple 7 (we do not have the latest version of Maple) where Maple allows one to compute high-precision Bessel functions by setting a global variable, `Digits`. Maple also allows one to view the source code for part of the implementation (sans comments) by `interface(verboseproc=2);`. Some detective work reveals that in this case it seems the code eventually calls `evalf/hypergeometric/kernel`, which is a built-in procedure without provided source code. Ultimately on `z40` converted to a 90-digit float, Maple provides 48 decimal digits correct, and 42 more, incorrect [5].

-.214544245178072877346184333649845274003117394888 *62...* E-40

If we repeat our experiment and move away from the zero, say `BesselJ(0,z40f+1/100);` then Maple 7 provides close to the requested 90 correct digits (actually 87).

Any of the algorithms we have encountered in programs for arbitrary precision Bessel functions seem capable of producing an arbitrary number of digits; however, (a) Some seem unaware of how many of the digits are correct for particular arguments and (b) we suspect that systems may choose algorithms that are far slower than they need be.

Since the precision specification for an unrestricted algorithm is revealed at run-time, the analysis and implementation of such algorithms is usually different from the traditional case of a pre-determined precision. The error analysis for any routine is aimed at finding an implementation that produces just-barely the number of digits required for any input.

The programmer for an ordinary library (fixed-precision) routine is given the bit representation and precision for numbers in advance, which is an advantage in designing a program to produce that number of bits *with the least amount of time and space.*

It is tempting to try to construct an unrestricted algorithm by using some convergent iteration until it is accurate enough, perhaps coordinated with some *a priori* bounds, as is done by MPFR. Unfortunately one cannot rely on the most tempting of conditions: assuming that if iterations n and $n+1$ agree in n digits, that those digits are correct. In fact such a proposed "converged" answer may not be correct at all, but an artifact of round-off. In other words, the computations do not agree, but this would only be revealed if (both) iterations were performed in higher precision.

The ACM Algorithm 236 for Bessel functions, which is otherwise perfectly adaptable to arbitrary-precision and uses the previously mentioned "backwards recurrence" [5] can exhibit this kind of false convergence near the zero-crossings of J_n. The computation can be boosted to higher precision *if one can tell that that is needed.* Interestingly, the version of Algorithm 236 directly translated to the computer algebra system Maxima can even give partially symbolic answers, e.g. $J_{7/3}(2)$ is approximately 690899216853/(582246954016*Γ(1/3)).

2 Taylor series near zero

For Bessel functions in the region close to the origin an easy and certainly *unrestricted* algorithm is one that starts with the Taylor series expansion at zero for J_0 and adds enough terms and enough precision so that the next non-zero remaining term does not affect the result. Near zero the successive terms drop substantially and they can be added in sequence. The coefficient of x^{2k} has magnitude $1/(4^k(k!)^2)$. (It seems to be slightly advantageous to use the Taylor series for $H(\nu,x) = J_\nu(\sqrt{x})$ which has all non-zero coefficients; $H(0,0.25)$ computes $J_0(0.5)$). Ten terms in this series is good for errors less than 10^{-21} for $x < 1.202$, which is half the distance to the first zero of J_0. If x is closer to that zero (at about 2.4048), it is more efficient to expand at that point. Setting aside that efficiency argument and neglecting the relative error at that first zero, the series seems like an unrestricted algorithm can add up the terms $x^k/(4^k(k!)^2)$ in $H(0,x)$ (of alternating sign) until they do not affect the answer.

As indicated earlier, summing this series suffers from numerical cancellation near a zero c of J_0 unless the expansion is about the point c.

[5] version 15 provides different though not more correct digits than version 7.

One can avoid cancellation by an even more expensive tactic, which is to use exact rational coefficients, accumulating the sum exactly until subsequent terms are below the specified threshold. This can require furious high-precision integer and rational number computation.

As indicated earlier, these efficiency concerns (many terms, high precision) legislate against expansions in a Taylor series about zero for large x, but do not necessarily detract when the series expansion is shifted so that it is centered on a nearby zero of J_0.

3 Asymptotic Formula, large x

For points far from $x = 0$ the computation of a Taylor series can be compared to the usual asymptotic formula (DLMF 10.17.3). This is presented later, andf shows that one can provide considerable accuracy with small effort. Unfortunately the asymptotic formula evaluated near a zero will also require computation in higher precision to achieve the same relative error.

The relevant DLMF formula 10.17.3 looks like this.

Where

$$a_k(\nu) = \frac{(4\nu^2 - 1^2)(4\nu^2 - 3^2)\cdots(4\nu^2 - (2k-1)^2)}{k!8^k},$$

and

$$\omega = z - \frac{1}{2}\nu\pi - \frac{1}{4}\pi,$$

then for ν fixed as $z \to \infty$,

$$J_\nu(z) \sim \left(\frac{2}{\pi z}\right)^{\frac{1}{2}} \left(\cos\omega \sum_{k=0}^{\infty}(-1)^k \frac{a_{2k}(\nu)}{z^{2k}} - \sin\omega \sum_{k=0}^{\infty}(-1)^k \frac{a_{2k+1}(\nu)}{z^{2k+1}}\right)$$

There is a question as to the bound on the accuracy of this asymptotic formula as a function of z. That is, for a relative small z as one includes more terms in the summations, a particular value is approached. It just might not be the required $p-$digit correct value, and no amount of higher precision $p+q$ arithmetic will change this.

There is a substantial published literature on Bessel functions in applied mathematics journals, a literature which is only briefly touched upon in our direct references. Each of our references has further bibliographic links. The classic volume by Watson [15] predates the explosion in computational power and consequent analysis unleashed upon many aspects of the Bessel functions and their applications. Some of the literature consists of theorems concerning Bessel function properties and/or properties of programs to compute related values. Some of the numerical literature includes recommendations for methods, but theoretical error bounds are generally inadequate in that they may not account for properties of computer arithmetic (finite representation, round-off, exponent overflow). These issues require additional analysis; we have found that testing can illuminate the occasionally over-conservative theoretical bounds.

Note that our program fragments, while executable, do not take advantage of every algorithmic tweak possible. Instead their central thrust is to indicate the efficient processes that are possible, and still retain some semblance of readability.

4 The 3-term Recurrence (Miller's algorithm)

Using a 3-term defining recurrence in reverse to compute values of Bessel functions is generally credited to to J.C.P. Miller [11]. There are a number of useful modifications to the basic idea as illustrated by Gautschi [5]. To see a relatively simple explanation, refer to section 9.12, Numerical Methods, Bessel Functions of Integer Order, [1], or the more recent online version of this publication which has a somewhat more abstract explanation, sections iii and iv of the Digital Library of Mathematical Functions (`http://dlmf.nist.gov/3.6`)

henceforth DLMF. These sections cover, respectively, Miller's algorithm and a variant, Olver's algorithm. [13]. A minimal version of Miller's backwards recurrence program is quite short, and is given below. It becomes more complicated if one imposes the (reasonable) additional requirement that the program determines (*a priori* a good value for the number of iterations (`nmax`, below) needed to meet some accuracy criterion [13]. In fact the program computes not just $J_0(x)$, but returns an array with values for all the integer-index Bessel functions up to $J_n(x)$, though the ones that matter must all be checked for suitable convergence, and the highest numbered ones will not be accurate. One of the tricky points in a program like this disappears if it can be executed in an arbitrary-precision floating-point setting where exponent overflow is avoided. The terms in the iteration can grow exponentially until they are divided by the normalization parameter λ in the formulation (`lam` in the program.) While there are other techniques to avoid this additional computation (cf.[17]), this is simply not a concern.

The 5-line program is based on the recurrence:

$$J_k(x) \;=\; 2(k+1)/x J_{k+1}(x) - J_{k+2}(x).$$

This is inadequate without a few additional observations, namely that for large enough k, $J_k(x)$ will be close to zero ($x > 0$) and $J_{k-1}(x)$ will be not quite so small: we set this arbitrarily to some value e.g. to 1, and correct it later. Running the recurrence backward to 0 we end up with values for $J_0, \cdots, J_n$ for $n < k-5$ or so, all of which are roughly a constant, call it λ, times their correct value. A value for λ can be computed by knowing that

$$\lambda = J_0(x) + \sum_{i=1}^{\infty} J_{2i}(x).$$

This factor can be estimated by summing the terms computed. Finally this factor can be divided out.

```
/* Bessel J functions. (the essential computation in ACM alg 236.)
   Returns a list of nmax items, approximate values for {J[0](x),J[1](x),..., J[nmax](x)}.
   Earlier items in the list are more accurate; the last is zero.  */

bj(x,nmax):=block([lam:0, r:[1,0],xi:1/x,li:0],
 for k:nmax-2 step -1 thru 0 do
   (push(2*(k+1)*xi*first(r)-second(r), r),
    if evenp(k) then lam:lam+first(r)),
  r/(2*lam-first(r)))
```

Here are some examples.

```
a:bj(3.5,40)$
```

`a[1]` is a value for $J_0(3.5) = -0.38012773998726$, accurate to all places shown. `a[21]` is $J_{20}(3.5)$ to about 14 decimals.

Since we are using a computer algebra system, there are some neat alternative uses of this simple program. For example, we can feed it a *symbolic* argument, and also indicate that the results should be simplified to rational (ratio-of-polynomial) forms. The command `r:bj(rat(x),10)` computes a list of 10 such forms. The value of `r[1]` is a function of arbitrary x which can be used in some range to compute $J_0(x)$. With `nmax=10` as above, a setting which assumes that $J_q(x)$ for $q > 10$ is negligible, the formula is:

$$\frac{5\,x^8 - 1920\,x^6 + 169344\,x^4 - 4128768\,x^2 + 18579456}{x^8 + 96\,x^6 + 8064\,x^4 + 516096\,x^2 + 18579456}.$$

To compute $J_0(40.0)$ to 40 decimal places, experimentation shows it is adequate to carry 45 decimal digits and compute `bj(40.0b0,117)[1]`. Implementing the decision process to figure these parameter settings for

$x = 40.0$ (that is, the floating point precision should be boosted to 45 or more and `nterms` should be 117 or more) adds somewhat to the 5-line program. Several methods are discussed by Olver and Sookne [13]. One cited method is to make a table to specify `nmax` for a range of x, order, and precision. Algorithm 236 [5] uses an approximation involving the polylogarithm. An idea credited to W. Kahan which only slightly over-estimates `nmax` seems to work well. On this example, it recommends `nmax=119` instead of 117. It is about 4 lines:

```
/* findn return a recommended value for nmax in the bj routine,
   given the argument x and d=number of decimal digits */
findn(x,d):=block([r:1, y:[1,0],lim:2*10^d],
  while abs(first(y))<lim do
    (push((2*(r-1)/x*first(y)-second(y)), y),     r:r+1),
  r-1)
```

This procedure estimates `nmax` by running the recurrence in the forward (unstable) direction to see how truncation errors accumulate. This information is used to predict how the backward recurrence will behave [13]. Caching values of this function (rounding `x` upward) may be worthwhile. It is trivial to do this in Maxima, though the cost (eventually) in storage may make it useful to be more clever, dropping out unused data which can anyway be recomputed.

```
findncached(x,d):= findnc[ceiling(x),d]; /* round up */
findnc[x,d]:=find(x,n)  /* a hashed array, "memoized" */
```

This method does not take any notice of the situation of difficult argument values, namely those near a zero of J_0. How good a job does `nmax=119` do for the argument z_{40} mentioned earlier, a number, which is about 40.05, and very near a root of J_0? In fact, $J_0(z_{40})$ is about -2.145E-41. As constructed so far, this procedure does not do very well; in spite of giving 40 correct digits at the nearby argument $x = 40.0$, at z_{40} the routine gives about 4 decimal digits correct! While it is a good answer in terms of absolute error, it is poor in terms of relative error. This is not inevitable. If we want a more accurate answer we can compensate for the 41 digits "lost" and use 41+45 digits of precision. In 86-digit arithmetic, all the initial 45 digits are correct! So in addition to Kahan's calculation for `nmax` we must also look at the answer: if the base-10 exponent is -d, we recompute with at least d more decimal digits[6]. A program to compute this compensation is in the next section.

How expensive is this recurrence? The iteration takes O(`nmax`) arithmetic operations. If the multiplications are executed in software arbitrary precision, then each one is considerably more expensive than the corresponding hardware floating-point operation. In the (probably unusual) case of excessive loss of precision we may have to re-run the computation (once).

We can save some time if we are only interested in one particular J_n, since we need not divide the whole list by λ. The program then looks like this:

```
/*return just j[n](x) */
bjn(x,n, nmax):=block([lam:0, r:[1,0],xi:1/x,li:0],
 for k:nmax-2 step -1 thru 0 do
   (push(2*(k+1)*xi*first(r)-second(r), r),
    if evenp(k) then lam:lam+first(r)),
  r[n+1]/(2*lam-first(r)))$
```

[6] We mention decimal digits for human consumption here. Of the systems we tested, only Maple actually uses a base which is a power of 10. The others are binary, which means we should talk about leading zero binary bits etc.

5 How to fix accuracy near a zero

If b is the answer from the first run of `bjn`, then we can compute the number of lost decimal digits as `r: ceiling(-log(abs(b))/log(10.0)-1)`. If `r` exceeds some acceptable level of relative error, 3 digits (say), we re-do the computation with higher floating-point precision: `fpprec:fpprec+r`. On return, round back to `fpprec`. It is advisable to boost the working precision by a few digits (say 3) prior to the initial run.

```
J0(x):=
 block([b,r,nmax,fpprec:fpprec+3],
  b:bjn(x,0,nmax: findn(x,fpprec-3)),
  r:ceiling(-log(abs(b))/log(10.0)-1),
  if r<3 then b
         else
        (b: block([fpprec:fpprec+r],
                  bjn(x,0,nmax)),
         bfloat(b))) /*round back to requested precision */
```

(The computation of `r` above can be avoided by simple arithmetic if one has an operation to extract the exponent.)

This works for any positive x but there is an efficiency issue for large arguments, where `nmax` gets rather large. For example, for $x = 1570.01$, near the 500th zero of J_0, we need 1706 terms just for 16 decimal digits. For such large values of x there are better ways, and we should include consideration of asymptotic formulas. These can be considered "unrestricted" in the sense that they can be boosted to have a small truncation error: there are expressions that assist in estimating the error in formulas. In section 10.17 in DLMF `http://dlmf.nist.gov/10.17` the formula 10.17.3 for $J_\nu(z)$ has the (slightly) helpful error bounds given in section 10.17(iii). (An equivalent formula occurs in Gradshteyn/Rhyzik [7] 8.451.1 p 961). The formula in question cannot be evaluated exactly in rational arithmetic but must be evaluated approximately. It includes several non-rational components[7] whose combination may result in significant numerical error.

```
/* Gradshteyn and Rhyzik 8.451.1 p 961, or DLMF 10.17 */

 (ww(n,m):=gamma(n+m+1/2)/(m!*gamma(n-m+1/2))/2^m,
   gg(n,lim,z,r):=sum((-1)^k*ww(n,2*k+r)/z^(2*k+r),k,0,lim),
j(n,z,lim,pi):=
  block([w: z-1/4*(2*n+1)*pi],
  sqrt(2/(pi*z))* (cos(w)*gg(n,lim,z,0)
                   -sin(w)*gg(n,lim,z,1))))$

/*alternatively, if m is an explicit non-negative integer, ww can be computed as in DLMF

 ww(n,m):= if m=0 then 1 else  (4*n^2-(2*m-1)^2)*ww(n,m-1)/(8*m)  */


/* This program can be used with SYMBOLIC inputs too, resulting in typesetable formulas, e.g. */
```

$$\frac{\sqrt{2}\left(\cos\left(x-\frac{\pi}{4}\right)\sum_{k=0}^{lim}\frac{(-1)^k\,\Gamma\left(2\,k+\frac{1}{2}\right)}{\Gamma\left(\frac{1}{2}-2\,k\right)2^{2\,k}\,(2\,k)!\,x^{2\,k}}-\sin\left(x-\frac{\pi}{4}\right)\sum_{k=0}^{lim}\frac{2^{-2\,k-1}\,(-1)^k\,\Gamma\left(2\,k+\frac{3}{2}\right)x^{-2\,k-1}}{\Gamma\left(-2\,k-\frac{1}{2}\right)(2\,k+1)!}\right)}{\sqrt{\pi x}}$$

[7] square-root, sin, cos, π

DLMF Section 10.17(iii) tells us that the remainder associated with each of the sum expressions does not exceed the first neglected term (beyond some initial stretch of about $\nu/2$ terms).

This is not comforting numerically. The two significant terms associated with sin and cos are nearly equal and opposite at the Bessel function zeros, since (in order to yield an answer near zero) they must nearly cancel out. The places where accurate digits are in short supply are exactly where the formula will have high relative error and therefore the formula, in spite of being "mathematically" accurate will return mostly noise digits unless computed to sufficiently higher precision to compensate. If the answer is required to p digits and the result has q leading zeros, then the subtraction (and essentially the rest of the computation) must be done in at least $p+q$ digit arithmetic to provide p correct ones. This is independent of the truncation error inherent in the mathematics of the formula caused by not carrying enough terms in the summations. Note that these sums are divergent and therefore must be computed only so far as justified, for example, by a test given near the end of this section.

In particular this asymptotic formula does not provide ever-increasing accuracy by merely bumping up the summation limits.

There is a way of rearranging the expressions (see, for example, the documentation for MPFR [12]). The idea is that instead of computing,

$$P(n,z)*\cos(w) - Q(n,z)*\sin(w)$$

there is an advantage in computing the mathematically equivalent

$$\frac{(-1)^{n/2}}{sqrt2}[P(n,z)(\sin z + \cos z) + Q(n,z)(\cos z - \sin z)]$$

for n even and

$$\frac{(-1)^{(n-1)/2}}{sqrt2}[P(n,z)(\sin z - \cos z) + Q(n,z)(\cos z + \sin z)]$$

for n odd.

This assumes that the error in computing $\sin(x)$ or $\cos(x)$ for large x say $x > 2^p$ in precision p is unacceptable but $\sin(x) + \cos(x)$ is less so. This does not address the concern that either formulation has a problem when the overall result is near zero. At such a point most of the significant figures in the sub-expressions cancel, and therefore the precision of the whole calculation must be boosted.

Considering $P(n,z)$ and $Q(n,z)$ together, the ratio of the terms of index $k+1$ over the term of index k is about $k/(2z)$. The series starts to diverge when $k > 2z$. At that point, the kth term is about e^{2z}, so for precision p, if $z > p/2\log 2$ the asymptotic formula is credible, so long as the result is not near a zero.

We turn to other approaches in the subsequent sections. First we find a mechanism to compute an important subpart of these other approaches, namely accurate and inexpensive derivatives.

6 Derivatives of Bessel Functions

We will need derivatives of Bessel functions at specified points for use in evaluation of Taylor series and Hermite interpolation, methods described later.

Here we describe how, as a consequence of various identities, the formula for each derivative of J_0 at its kth zero z_k can be rephrased entirely as a rational function of z_k multiplied by $J_1(z_k)$. (See equation (9) in Wills [16].) Compared to Wills, the formulas become even simpler because at our points of interest we know that $J_0(z_k) = 0$. We combine this computation with dividing the nth derivative by $n!$:

$$J_0(x) = 0; \quad J_0'(x) = -J_1(x); \quad J_0''(x)/2 = J_1(x)/(2x)$$

$$\frac{J_0^{(k)}(x)}{k!} = \frac{1}{x}\big(\frac{1-k}{k}J_0^{(k-1)} - \frac{1}{k(k-1)}J_0^{(k-3)}(x)\big) - \frac{1}{k(k-1)}J_0^{(k-2)}(x).$$

It might seem at first that reducing the evaluation of $J_0(x)$ to evaluation of $J_1(z_k)$ is not much of an advantage, but note that we are accessing the value of J_1 only at *pre-computed zeros of* J_0, not at x. Furthermore, while the fully-accurate direct evaluation of J_0 near its zeros is delicate, such points do not represent difficulty for evaluating J_1.

As an example, the first six values for $1/k!(d^k/dz^k)J_0(z)$ where $J_0(z) = 0$ are *exactly*

$$\left[-1, \frac{1}{2\,z}, \frac{z^2-2}{6\,z^2}, -\frac{z^2-3}{12\,z^3}, -\frac{z^4-7\,z^2+24}{120\,z^4}, \frac{z^4-11\,z^2+40}{240\,z^5}\right]$$

times $J_1(z)$. There are many ways of arranging these formulas. For example, if we first set $y = 1/z$ we can state those formulas using only polynomials with rational constants (here we omit division by $n!$) as

$$\left[-1, y,\ 1-2\,y^2, 6\,y^3-2\,y,\ -24\,y^4+7\ y^2-1,\ 120\,y^5-33\,y^3+3\,y\right].$$

Another rearrangement would be using Horner's rule. If we are dealing with simple numbers, we need not worry about "expression swell" and not be too concerned about roundoff, and so can use the recurrence above as shown in the Maxima program below.

```
/*[J0(x), J0'(x), J0''(x)/2, ..., J0{n}(x)/n!].
We require as input n, x, and a sufficiently accurate value for J1(x) */

J0Ders(n,x,J1x):=  /* all the derivatives  of J0 at x*/
 block([xinv:1/x,  ans:[1/(2*x),-1,0]],
   for k:2 thru n-1 do
     push(-((ans[1]*k^2+ans[3])*xinv+ans[2])/(k^2+k), ans),
  reverse(J1x*ans))
```

Note that the value of the nth derivative is a simple function of the values of the $n-1$, $n-2$, and $n-3$ derivatives, so the algebra for making and displaying the explicit formula is probably unnecessary. We can compute numeric terms as needed, 4 multiplies and 3 adds. A convenient data structure is a list rather than an array; each additional derivative refers to only the top 3 elements. Also note that in Maxima the type of the arithmetic is based on the type of the values of `x` and `J1x`. If either of these is a machine float, the whole calculation will be done in machine floats. If neither is a machine float but either is a bigfloat, the whole calculation will be done in bigfloats of precision equal to the global floating-point precision. If both are rational numbers or integers, the entire calculation will be done exactly. If one or both are symbolic, the entire calculation will be symbolic. If x is a rational form, the result will be in canonical simplified rational form.

We are not quite done with this program unless we can determine that the answers it produces are accurate: Is the sequence, as computed, stable? If we need k terms accurate to d decimal digits, how many digits of $J_1(x)$ do we need, and what should the working precision be?

Experiments suggest that at zeros near $z = 100.0$ or 1000 or even near 10,000, using ordinary machine double-floats provide coefficients that are good for about 16 decimal digits all the way out to $n = 100$, although the coefficients underflow to zero before that point. Using arbitrary precision software set at 25-decimal precision, roundoff eventually degrades the results after about 150 coefficients (when the coefficients look like 10^{-264}). We suspect people do not really need 264 digit Bessel values, but there you are.

One variation in the use of this program will produce a result that has either zero or one rounding, and so can be as precise as the input allows. If `x` and `J1x` are given as rationals, (approximation to the presumably

irrational value of J1x) the answer will be a list of rationals, and there will be no roundoff at all. The numbers will probably be presented with uncomfortably large numerators and denominators. If only x is given as a rational, the bulk of the calculation will be done exactly, and then there will be one rounding when multiplying by J1x which is presumably a bigfloat or float. Alternatively, binding J1x to 1 and subsequently multiplying the result of the summation by $J_1(x)$ would work. This modification essentially removes the arithmetic error, but would be time-consuming so we would not want to do this except at "table-building" time.

While appealing, this does not eliminate *all* sources of error: since x is a rational number it cannot be exactly a root of $J_0(x) = 0$ unless $x = 0$, and so we are computing something subtlely different from our goal. We can finesse this problem if we know that the chosen rational value for x is so accurate that the residual $\epsilon = J_0(x)$ is far less than $10^{-d} J_1(x)$, where d is the bigfloat precision. We should also assure ourselves that the rational iteration in the derivative program is not especially sensitive to slight changes in the value of x. Fortunately we know that $f(x + \epsilon)$ can be approximated by $f(x) + \epsilon f'(x)$, and we have approximations of $f'(x)$ easily available in the list we are computing! Thus we can multiply the nth normalized derivative by n to give the rate of change with error of the previous normalized derivative. If the (rational) value for x is known to be good to d decimal digits, we have a value for $\epsilon = 1/10^d$. Note that this could be used with bigfloats (of sufficiently high precision) but it certainly works because computing with a rational value for x removes all round-off error!

```
/*[J0(x), J0'(x), J0''(x)/2, ..., J0{n}(x)/n!].
We require as input n, x, and J1(x). One or zero roundoff..  */

J0DersExact(n,x,J1x):=  /* all the derivatives  of J0 at x*/
 block([xi:1/x, ans:[1/(2*x),-1,0]],
   for k:3 thru n do
     push(xi*((1-k)/k*first(ans) -1/(k*(k-1))*third(ans))
                          -second(ans)/(k*(k-1)),
           ans),
   reverse(ans) *J1x )


/* error in nth deriv in list m; m[1]= function, m[k] = (k-1)th deriv/(k-1)!,
   eps = known bound of error in x, a rational approximation to a zero of J_0 */

error_in_der(x,eps,m,n):= eps*m[n]/n
```

7 Taylor series in summary

Given a list of coefficients appropriate for a Taylor series for f namely $\{f(x_0),\ f'(x_0),\ f''(x_0)/2, ...,\ f^{(n)}(x_0)/n!\}$, here is a Taylor series evaluation program for $f(v)$:

```
(tayser(m,len,x0,v):=
  block([q:v-x0],
    tayser1(m,index):= if index>len then 0
                else first(m)+q*tayser1(rest(m),index+1),
  tayser1(m,1)))
```

This program adds the small terms together first. To use this program for Bessel functions we can use a zero for x0 and for m, a list of coefficients from the derivative program of the previous section.

It might be helpful to see how rapidly the coefficients drop.

Below we have printed the first 5 digits of the coefficients around the zero at c0=2.404825557695770 to demonstrate how many terms in the sum are likely to be needed. (These coefficients will be multiplied by $(x - x_0)^n$). The 40th coefficient is approximately 1.1×10^{-32}.

```
[ 0,          -0.519,       1.0793E-1,     5.6601E-2,     -8.6576E-3,
 -2.1942E-3,   2.6437E-4,   4.3729E-5,    -4.3388E-6,     -5.3049E-7,
  4.4700E-8,   4.3265E-9,  -

3.1666E-10,   -2.5338E-11,     1.6387E-12,
  1.1169E-13, -6.4695E-15, -3.8392E-16,    2.0132E-17,     1.0576E-18,
 -5.0661E-20, -2.3875E-21,  1.0538E-22,    4.4341E-24,    -1.5938E-25,
 -1.6803E-26,  4.1271E-27, -1.5342E-27,    6.1900E-28,    -2.4874E-28,
  1.0001E-28, -4.0255E-29,  1.6219E-29,   -6.5415E-30,     2.6406E-30,
 -1.0668E-30,  4.3137E-31, -1.7455E-31,    7.0683E-32,    -2.8641E-32,
  1.1613E-32].
```

Around the 100th zero at about 313.37 the coefficients drop faster; the 40th coefficient is then about 3.49E-51.

As shown in the previous section on derivatives, the coefficients could be stored as formulas, or computed as needed by the program `J0Ders` or stored for some set of zeros. If more precision is needed, or expansion around a zero larger than had been anticipated is needed, the coefficients can be recomputed and stored (again). An accurate value of $J_1(c)$ is also needed: this can be computed by the recurrence program. As one proceeds down the sum, the coefficients toward the end need not be computed or stored or operated on to full precision. If we have arranged to sum the terms from the smallest to the largest, a 50-digit correct result can be obtained even if the initial 10 terms are added together in double-float precision; only the last 10 terms or so require 50-digit arithmetic.

There is no penalty in asking for the value at a large argument if the data on a nearby zero has been pre-tabulated. There is still an issue of accurate computation (lost digits) that has to be treated in the same way as the previous section, and the number of terms to be used can be computed at runtime. A program that adds terms "until it doesn't matter" is a bit tricky: the coefficients may not be strictly monotonically decreasing in absolute value. Encountering a zero coefficient doesn't mean that all subsequent terms are zero. Instead one can use an *a priori* error bound on the Taylor series. The Taylor series computation could require higher precision in the coefficient list, as well as more terms in that list to meet a required relative error bound.

```
(tayser2(m,x0,v,precision_goal):=
[[ messy details omitted ... call tayser() with estimated length;
  call tayser() with more terms and higher precision;
  if they agree to precision_goal, return the value]]
)
```

Remaining to be seen are two comparisons. For arguments that are large enough and precision requirements that are modest, is the Taylor series just too much work? Reasonable alternatives are Hermite Interpolation and asymptotic series.

8 Hermite Interpolation

There are many variations on techniques for interpolation, where some information about function values at existing points is used to determine or approximate function values at additional points. For example, linear

interpolation assumes that a function f varies approximately linearly between two tabulated points x_0 and x_1 thus the value of f at an intermediate point lies on the straight line between $(x_0, f(x_0))$ and $(x_1, f(x_1))$. More elaborate interpolation using more points fits a more elaborate curve to the tabulated points usually resulting in a more accurate result. Hermite interpolation differs in that derivatives of f are also given at some or all of the points. The spacing may not be even. In our case we are mostly going to look at Hermite interpolation with exactly 2 points, but since we have neat ways of computing derivatives, we exploit this information.

Since Hermite interpolation may be less familar than (say) Taylor series, we discuss it informally first.

Consider this problem: I'm thinking of a function f.

I know its derivatives at zero. Here are the first 7. (The zeroth derivative is $f(0)$.) In Maxima, we can compute the list of derivatives this way:

```
s0:  makelist(at(diff(f(x),x,n),x=0),n,0,8);
```

which returns`[0,1,0,-1,0,1,0,-1,0]`.

Similarly, I know its derivative at π.

```
spi:  makelist(at(diff(f(x),x,n),x=%pi),n,0,8);
```

returns `[0,-1,0,1,0,-1,0,1,0]` What is the value of $f(1/2)$ approximately?[8]

I can incorporate the values at 0 and π, and 4 derivatives. (or any number of derivatives) into a calculation, via Hermite Interpolation, to estimate the value of $f(q)$.

The 2-point Hermite interpolation function `hi` (defined below) has the expected property that given this data the error vanishes at 0 and pi. It also uses the derivatives at two points. The approximation can be improved by using more derivatives and does not depend on knowledge of function values at new intermediate points.

```
hi(s0,spi,9,0,bfloat(%pi),1.5b0)
```

returns 0.99749498660*35*.. while `sin(1.5b0)` returns 0.99749498660405... [9]

It is natural to wonder how efficient, in terms of error, Hermite interpolation might be. If you use a total of 10 data items from each of two points the error bound is similar to that of a Taylor series of 20 terms: proportional to (1/21!) times an 11th derivative evaluated at some point in the interval.

Our scheme then is, given a point of interest v to choose two nearby points: those zeros of the Bessel function J_0 which bracket v. If the point v is "very" close to one of the zeros we can actually do better with the Taylor series. We would need to quantify "very" though.

If we use Hermite interpolation, we find it is easy (i.e. in working precision that is modest) to evaluate $J_0(v)$ accurately (i.e. with low relative error) even in the places where $J_0(x)$ is difficult (i.e. requires much higher working precision) to evaluate by recurrence. Hermite interpolation is a good choice for places spaced somewhat mid-way between zeros of J_0.

If we are given some modest restriction, say that we will need no more than 100 decimal digits of numerical values, and arguments less than, say, 300, we can tabulate the first (say) 100 zeros of J_0 and prospectively evaluate the needed derivatives, perhaps caching them. Given z, some zero of J_0, $d^n/dx^2(J_0(x))|_{x=z}$ is a *simple rational function* of z times $J_1(z)$. In fact, that rational function is a polynomial divided by z^n. These polynomial functions are easily tabulated, and are not delicate to compute at those points z. This is convenient since these coefficients are good irrespective of future perhaps more demanding precision requirements.

There are other types of interpolations. For those familiar with the technology we note: the Chebyshev approximation will not match the endpoints, which we really would like to see if we are going to maintain low

[8]If you have guessed that the function looks like $\sin(x)$ I won't deny it.

[9]We use the convention that digits that are not correct are typeset in italics.

relative error near zeros. (We can match the endpoints in the Chebyshev approximation, compromising the minimax property). The Hermite approximation will not be so uniformly close, but will match the endpoints and also be rather good beyond them. As noted, Taylor series (of the same degree) will be even better than the Hermite approximation sufficiently close to the point of expansion, becoming much worse further away.

For comparison we have tried each on the expression $\sin(\pi x)$ between -1 and 1. (see fHI, fCHI, fTS below).

A simple way of computing an Hermite Interpolation formula is via an implementation of divided differences. This takes only a few lines of code in Maxima. The program assumes two points, two lists of normalized derivatives, where the nth derivative is divided by $n!$:

```
 /*hermite interp.at 2 points, use len-1 derivs.
 m0 and m1 are each available lists of normalized derivatives at x0 and x1.
 (nth derivative divided by n!)
   Evaluate the function at v which can be any numeric type or symbolic. */


hi(m0,m1,len,x0,x1,v)   :=
 block([del:x1-x0 ,v0:v-x0,v1:v-x1],
  /* Compute a divided difference tableau using derivatives.*/
 kill(dd),
 dd[K,L]:= if K=0 then m1[L] else if L=0 then m0[K] else
            (dd[K-1,L]-dd[K,L-1])/del,

/* auxiliary function for hermite interpolation, start it at hi2(1,0),
   This computes the Newton interpolation polynomial.  It
   refers to external values for dd[], m0, m1, len ,v0 and v1*/

 hi2(K,L):= dd[K,L]+
  (if K=L then 0
          else if (K<len) then v0*hi2(K+1,L)
                          else  if (K=len and L=0)  then v0*hi2(K,L+1)
                                                    else v1*hi2(K,L+1)),
 hi2(1,0))
```

Since we anticipate not necessarily being content with a single result without some indication of actual error, here's another version that returns two approximations that can be compared for convergence. The advantage here is that some intermediate data can be easily re-used. Some of it could also be cached external to the program.

```
 /* Similar to hi except that TWO approximations are returned.
    As long as we have the DD tableau computed, see how much
    difference it makes if we do not use it all. The first of
    the pair of returned values is the same as above. The second
    ignores one derivative at each point and thus should be
    less accurate.  This makes it easy to test to see if the extra derivatives
     matter.  Note that the divided differences are
    already computed and stored from the first run. */
```

```
hiTWO(m0,m1,len,x0,x1,v)  :=
 block([del:x1-x0,v0:v-x0,v1:v-x1],
  /* Compute a divided difference tableau using
     derivatives when given and needed. Cache values.*/
 kill(dd),
 dd[K,L]:= if K=0 then m1[L]/ else if L=0 then m0[K] else
           (dd[K-1,L]-dd[K,L-1])/del,
 hi2(K,L):= dd[K,L]+
  (if K=L then 0
         else if (K<len) then v0*hi2(K+1,L)
                         else  if (K=len and L=0)  then v0*hi2(K,L+1)
                                                   else v1*hi2(K,L+1)),

 [ hi2(1,0), (len:len-1),hi2(1,0)]);
```

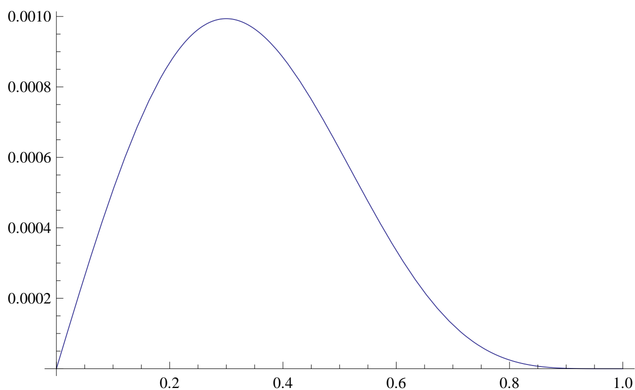
The Hermite approximation to $\sin(x)$ can be compared to a similar-cost Chebyshev approximation, computed via a library routine e.g.

```
 load("c:/cheby/cheby.mac")
ToCheby2(lambda([x],sin(npi*x)),7)
f2(x):=''FromCheb(%,x);
```

We compare various polynomials attempting to approximate $\sin(\pi x)$. We subtract the approximations from $\sin(\pi x)$, for x in $[0, 1]$ below. It is important to note the scales.

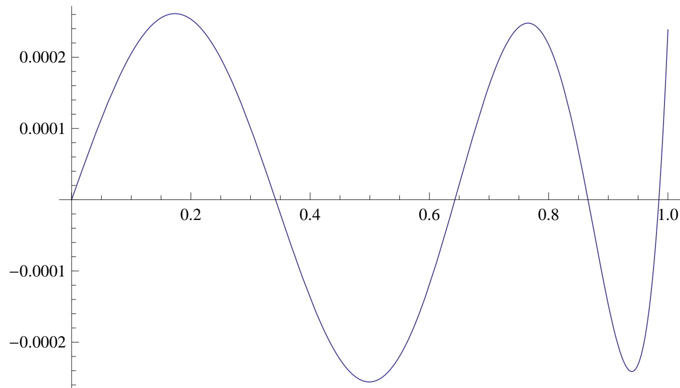
First is an Hermite Interpolation error curve.

```
fHI(x):=
sin(3.141592653589793*x)-0.051574119333151*x^9
                               +0.54208008407317*x^7
                               -2.494892781316772*x^5+5.140638114541538*x^3
                                -3.136251297964785*x
```



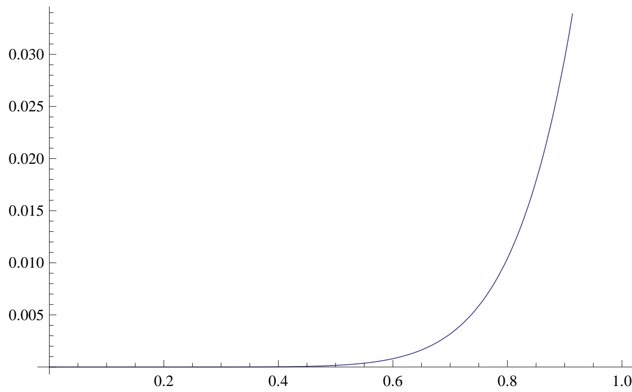
The error for Chebyshev error curve ..

```
fCH(x):= sin(3.141592653589793*x)+
                               +0.43742613047569*x^7
                               -2.434012102516614*x^5
                               +5.136060016937784*x^3
                               -3.139235550083358*x
```



The Taylor series

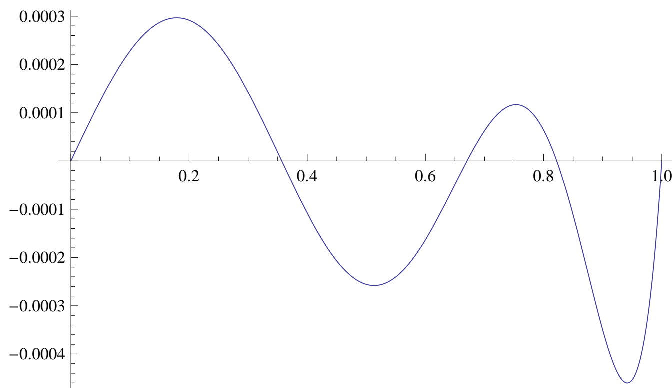
```
fTS(x):=sin(3.141592653589793*x)
          -(-0.59926452932079*x^7+2.550164039877345*x^5-5.167712780049969*x^3
                                   +3.141592653589793*x)
```



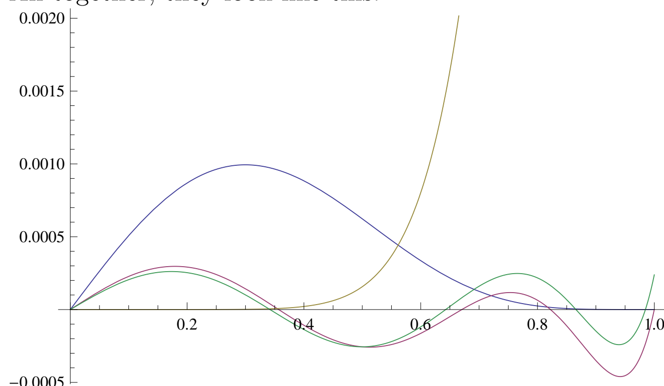
The accuracy of the Taylor series is far better than the others near the expansion point. The error at 0.2 is about 4E-8; even at 0.5 it is 1.56E-4.

and finally the Chebyshev approximation altered so that it achieves exact match at the endpoints -1 and 1, but consequently higher errors elsewhere.

```
fCHE(x):= sin(3.141592653589793*x)+0.43719514757848*x^7-2.433093975923623*x^5
                                  +5.134897514313284*x^3-3.138998685968137*x
```



All together, they look like this:



We can put Bessel derivatives into a Taylor series or an Hermite interpolation formula to compute $J_0(x)$ given two nearby zeros c_1 and c_2. We need high-precision values for c_1, c_2, $J_1(c_1)$ and $J1(c_2)$, (all presumably tabulated or computed on demand and then tabulated). Saving the values makes sense if one assumes that a likely use of these routine would involving computing Bessel functions repeatedly in some region.

There are a number of details that need to be resolved. These include finding the nearest zero points bracketing the argument, choosing the parameter `lim` (how many terms to try first) and how much extra precision (if any) is needed in computing the formula to achieve the desired accuracy. As discussed in Appendix 2, the coefficients need not all be of the same precision.

9 Example

Say we want to find the value of $J_0(x)$ for test values $p_1 = 31$, $p_2 = 33.77581$ to high precision. Note that there is a zero of J_0 at $c_1 = 30.63460646843198$. The next higher zero is at $c_2 = 33.77582021357357$. The backwards-recursion method easily computes $J_0(p_1)$ since it is separated from a zero. However, p_2 needs more work to achieve a given relative error.

[Note that the location of the zeros is pretabulated. We could choose a higher precision] Observe that $c_2 - c_1 = 3.1412..$; asymptotically the difference between successive zeros will be π.

We also observe that $J_1(c_1) = -0.14416597768637$ and $J_1(c_2) = 0.13729694340850296$ approximately.

A few of the (normalized) derivatives at c_1 (without the $J_1(c_1)$ factor) look like this:

```
[0,-1,0.01632141090225,0.16631148206155,-0.0027115394795455,
-0.008271403108424,1.3442372833683095E-4,1.9528448798324826E-4,
-3.1567676959633588E-6,-2.68163230902058E-6,4.3028340531506672E-8]
```

and similarly at c_2,

```
 [0,-1,0.014803489503389,0.16637447559803,-0.0024607600794386,
-0.0082823535724391,1.2217670605973137E-4,1.9583302500667524E-4,
-2.8761504725702065E-6,-2.6944506121076784E-6,3.9331824129655546E-8]
```

Note that in the Hermite formula we will be using $(x - c_i)$ terms, and $c_1 \leq x \leq c_2$, an interval of width approximately π.

Using only a subset of these 10 coefficients in the interpolation with an accurate value for $(x-c_1)$, $(x-c_2)$ as well as $J_1(c_1)$ and $J_1(c_2)$ we get, using 5 coefficients: 0.0512081*082284273*, and for 6: 0.051208145*019708*, for 9: 0.051208145304542, which is correct to all digits.

We have used the convention of displaying inaccurate digits in italics.

All the computations can be done in any arbitrary precision given only that there is sufficient accuracy in the values of c_i and $J_1(c_i)$.

How about values close to zeros?

Using 4 (or more!) terms we get the value of $J_0(3063461/100000)$ to be 5.091319*2692870853*E-7, but in fact this fails to be accurate, and using more interpolation terms does not change this result. To get higher accuracy we need more precise data for derivatives etc, *not more terms!*.

If we boost $c_1, c_2, J_1(c_1), J_1(c_2)$ to 40 decimal places we get a value of 5.091319277256023745..E-7 whereas the Maxima `bessel_j` program gives 5.091319274*011384* E-7, and the Mathematica 8.0.4.0 program gives 5.091319275*185995* E-7. However, Mathematica running in 40 digits gives a result that agrees with our boosted result.

```
try again with fpprec:40;
c1:    30.63460646843197511754957892685423273728b0;
j1c1: -0.1441659776863732076427180372796312563910b0;
c2:    33.77582021357356868423854634671472802393b0;
j1c2:  0.1372969434085029724559526482537179062473b0;
m1:makelist(H[i](c1)*j1c1,i,0,10);
m2:makelist(H[i](c2)*j1c2,i,0,10);
```

Let's kick this up a notch.. Mathematica's $J_0(3063461/100000)$ to 40 digits is accurately computed as:

5.091319277256023745310524125606674209141E-7.

We get 5.0913192772560237453105241256066742091405514427055474091700072*28...*b-7 120 recurrence terms and more digits for more (160 total) steps:

5.091319277256023745310524125606674209140551442705547409170007198756584682843158684867666749717*555791*b-7

But our interpolation gives

5.091319277256023745310524125606*64418499*b-7

5.091319277256023745310524125606674209140551442*691972876*b-7, fpprec:100 in maxima. 8 terms

5.091319277256023745310524125606674209140551442705547409170007198756584*352...*1b-7, 11 terms

From these results we see that computing with 40 decimal digit constants is not enough for 40 digits in the answer.

```
fpprec:100;
c1:    30.63460646843197511754957892685423273727357162917814719075501789971602444755672302323011597795469
j1c1:-0.1441659776863732076427180372796312563910822573751374043395193632465853862076336747643805364981350
c2:    33.77582021357356868423854634671472802393270574256681654896409701271934118168846761786101715399838
j1c2: 0.1372969434085029724559526482537179062473271785326411462547475443951121853459076527712805453021340
m1:makelist(H[i](c1)*j1c1,i,0,10);
m2:makelist(H[i](c2)*j1c2,i,0,10);
hiTWO(m1,m2,8,c1,c2,(3063461/100000));
```

This gives us 5.091319277256023745310524125606674209140551442705547409170007198756584*3524...* b-7, using only 11 terms, 70 decimal digits correct.

What about Taylor series? If we try 11 terms here:

`tayser(m1,11,c1,3063461/100000.0b0)` gives

5.091319277256023745310524125606674209140551442705547409170007*2356...*b-7, 61 digits correct 20 gives..

5.091319277256023745310524125606674209140551442705547409170007198756584682843158684867666749714*458...*b-7, 93 digits correct

What next? Let's try somewhat away from the endpoint: at the point 31, using 11 derivatives in 100 dec. digit arithmetic, we get just

0.05120814530454224879*96975...* from Maxima. (more accurate versions appear below)

This illustrates that far from the expansion point for the Taylor series, convergence is relatively slower. Recall that we are multiplying coefficients by $(x - v)^n$, a term that can be as much as $(\pi/2)^n$.

In these cases the recurrence backward is more promising. 90 steps of backward recurrence gets this:

0.05120814530454224879982049104889783... and 160 steps gives a number correct to 94 decimal digits. 5.09131927725602374531052412560667420914055144270554740917000719875658468284315868486766674971*7604*E-7

A combination of programs seems to have advantages where the rather rapid convergence of the backward recurrence is generally used at some distance from the zeros, and for modest argument.

An alternative to the Hermite interpolation is to just use a Taylor series for REALLY close values to zeros.

For the smallest positive zero of J_0, near 2.404, the coefficients drop slower than we'd like: the 10th coefficient is about 10^{-7}; it takes 100 terms to get down to 10^{-44}.

Even after division by the $k!$ denominator, the coefficients in the Taylor series drop modestly if we need 100s of digits.

In summary then, we can precompute and tabulate special values, or as is sometimes done, cache values of coefficients as they are first computed. If necessary, we can increase the precision of cached values (those that are not exact rationals) as a computation proceeds, assuming that Bessel functions are computed in "nearby" bunches and that once a given precision is required, there will be further requirements at that precision.

10 Finding zeros of BesselJ

There is a wealth of material on approximations for the zeros of Bessel functions. A 1996 paper by Lether [9] cites earlier work and also provides a rational formula for approximating the nth zero to 13 decimal digits or more. A 1991 paper by Ikebe *et al* [8] reports on an implementation of a method (see also Ball, [2]) of finding Bessel zeros from the eigenvalues of an infinite (but truncated) tridiagonal positive definite matrix. Since one cannot find all the eigenvalues of an infinite matrix, this paper includes a formula for a bound on the error from a given truncation of matrix size. Without testing for speed, it is clearly an advantage of this approach that a collection of zeros (up to some level of acceptability) are produced simultaneously.

In order to try this out, we have taken the code for this task from LAPACK (originally from EISPACK) and modified it to run using arbitrary precision floats from MPFR. In particular, the routine `Bisect`, which can refine an eigenvalue in a specified interval to arbitrary accuracy, given enough precision, seems to be the best option. Complicating the situation is that the true zero is an eigenvalue of an infinite matrix, and we must truncate it as a practical matter, approaching the desired eigenvalue incrementally.

We have seen that – with some boost to the working precision near zeros – we can find an accurate value of J_0. Given that we can tabulate reasonable zeros to any desired precision, and we can extend our table of zeros to other zeros as needed, we can cache values at first need. We work under the assumption that these values may be of interest in subsequent computations of $J_0(x)$ for nearby values of x.

Here is a function that computes a substantially accurate formula for the value of the nth positive zero of J_0, including a good value for the first few zeros. If this is of insufficient accuracy, we can (a) use a Newton iteration and find some more digits, or (b) use the eigenvalue method and simultaneously (and at considerable expense) upgrade all the tabulated zeros up to some value. The cost of computing zeros would be amortized over some number of calls, incurred only when addition information about zeros is needed: either higher accuracy of tabulated zeros or location of new zeros. These can be computed with methods given here, even if not so rapidly.[10]

[10] We are tempted to suggest that very high precision values of many zeros can be stored "in the cloud" from which they can be retrieved for local consumption when necessary.

```
jz2(n,pi):=block([b:(n-1/4)*pi,bi:0], bi:1/b, v:bi^2,
b+bi*(1567450796/12539606369+8903660/2365861*v+10747040/536751*v^2+
17590991/1696654*v^3)/ (1+
29354255/954518*v+76900001/431847*v^2+67237052/442411*v^3))
```

In some sense a more interesting simple formula whose accuracy increases with n but is less accurate for $n < 6$ or so is this:

```
jzero(n,pi):=block([c:1/((4*n-1)*pi)], if n<6
then jz2(n,pi) else (n-1/4)*pi +c/2-31/6*c^3 +3779/15*c^5
-6277237/210*c^7 + 2092163573/315*c^9 -8249725736393/3465*c^11 +
847496887251128654/675675*c^13)$
```

(For n=3, this formula delivers only 8 digits; for n=4000, it delivers 40 digits.)

In each case we feed in a numeric (rational, float, or bigfloat) value for π, whose accuracy will of course affect the computation. If only bigfloat values are needed for the zeros we could always call `jzero(n,bfloat(%pi))$` and the current global value of the floating point precision will be used.

While the goal in the "classical" numerical analysis approach seems to be to find the best formula, it is clear that the latter formula can simply be modified with a brief table to catch the requests for smaller zeros. Here we show rational approximations.

```
jzeroX(n,pi):=block([c=1/((4*n-1)*pi)],
if n<6 then
[ 17203146289/7153594253, 19873019651/3600133776,
26375592587/3047887899, 28886417038/2449758951,
19641207599/1315472229][n] else jzero(n,pi)
```

If these initial guesses are inadequate, there is a simple formulation, via Newton iteration, to improve the estimates. A more careful analysis and an implementation in MPFR arithmetic uses increasing precision for the steps in the calculation, where each step refines the estimate.

```
/* Newton iteration to a zero of J0, starting with a good approximation, h */
nextz(h,order):=block([fd:bj(h,order)], h+fd[1]/fd[2]);
```

Note that `bj[2]` is $J_1(x)$; the derivative is $-J1(x)$ hence the "+" rather than "-" in the Newton iteration.

We can compute

```
h:jzero(n,bfloat(%pi))
```

to start, and then `nextz(h,20)` until 2 iterations are insubstantially different, then `nextz(h,40)` and see if that differs; else repeat until 2 iterations are insubstantially different, then boost the precision, etc. (See `mpfr.lisp` file in our generic lisp arithmetic package available onine for program details.)

11 Conclusion

We've indicated the principles on which unrestricted algorithms for computing Bessel functions J_n, but especially J_0 can be designed. The illustrations using computer algebra system programs benefit from the easy availability of variable-precision arithmetic and the potential use of symbolic manipulation for gaining insight into the meaning of the generated computational schemes. Difficulties encountered in using the backward recurrence formula can be alleviated in part by using different programs which use information of derivatives at zeros of J_0 to find relatively quick and accurate values at large arguments.

It may be futile to try to optimize the choice between Taylor and Hermite algorithms: If the point in question is really close (just how to quantify is unclear) to a zero, a Taylor series is really good.

If the point is (say) halfway between two zeros, the question is, which is more work (including needed high-precision accuracy arithmetic): a Hermite interpolation using 10 data points at 2 zeros, or 20 data points at one zero, or use of the 3-term recurrence. Assuming there is a break-even point in computational effort between Taylor and Hermite, where is it, and are the expected properties preserved. (I'm assuming that the 3-term recurrence is in fact more work most places, though convenient if you want the answer with one method.)

Methods embedded in computer systems with variable-precision arithmetic can, and probably do, solve accuracy problems by (perhaps extravagant) boosting of precision and testing for convergence.

References

[1] M. Abramowitz, I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Mathematics Series 55, 10th printing, Dec. 1972. `http://www.e-booksdirectory.com/details.php?ebook=2781` updated version is the DLMF `http://dlmf.nist.gov/10.17`.

[2] James Ball, "Automatic Computation of Zeros of Bessel Functions and Other Special Functions", SIAM J. Sci. Comput. 21, pp. 1458-1464. (2000)

[3] R. P. Brent, "Unrestricted Algorithms for Elementary and Special Functions," http://arxiv.org/abs/1004.3621

[4] R.M. Corless, D.J. Jeffrey, H. Rasmussen, "Numerical evaluation of airy functions with complex arguments," *J. of Computational Physics Vol. 99*, no. 1, March 1992, 106114.

[5] W. Gautschi, Algorithm 236, "Bessel functions of the first kind," *Comm. ACM 7* no. 8 Aug. 1964, 479-480.

[6] Walter Gautschi. "Computational Aspects of Three-Term Recurrence Relations." *SIAM Review, 9*(1) Jan. 1967, 2480.

[7] Gradshteyn, I.S. and and Rhyzik, I. M., *Table of Integrals, Series, and Products,* Academic Press, 6th printing, 1980.

[8] Yasuhiko Ikebe, Yasushi Kikuchi and Issei Fujishiro, "Computing zeros and orders of Bessel functions" *J. of Computational and Appl. Math. 38* (1991) 169-184.

[9] Frank G. Lether, "Rational approximation formulas for computing the positive zeros of $J_0(x)$," *J. Computational and Appl. Math. 67* (1996) 167-172.

[10] Marc Mezzarobba. "NumGfun: a Package for Numerical and Analytic Computation with D-finite Functions." *Proc. 2010 ISSAC* (arXiv:1002.3077, doi:10.1145/1837934.1837965.

[11] J.C.P. Miller, *British Association for the Advancement of Science Mathematical Tables. Vol.X. Bessel Functions. Part II, Functions of Positive Integer Order,* Cambridge University Press, Cambridge, 1952.

[12] The MPFR Team, "The MPFR Library: Algorithms and Proofs", www.mpfr.org/algorithms.pdf, last visited 7/2012.

[13] F. W. J. Olver and D. J. Sookne, "Note on Backward Recurrence Algorithms," *Math. Comp.26,* No. 120 (Oct., 1972), 941-947.

[14] Henry C. Thacher, Jr. "New Backward Recurrences for Bessel Functions," *Math. Comp. vol. 22* no. 146, Apr. 1979, 744–764.

[15] G.N. Watson, *A treatise on the theory of Bessel functions*, Cambridge Mathematical Library, Cambridge University Press, 1944.

[16] C. A. Wills, J. M. Blair and P. L. Ragde, "Rational Chebyshev Approximations for the Bessel Functions $J_0(x), J_1(x), Y_0(x), Y_1(x)$", *Math. Comp, 39,* No. 160, Oct., 1982 , 617–643. http://www.jstor.org/stable/2007338.

[17] A. V. Zaitsev, "Implementation of Miller's method for evaluation of Bessel functions of first kind," *J. of Mathematical Sciences 63* no. 5 (1993), 558-560.