Complex Numerical Values of the Wright ω function

Robert M. Corless and David J. Jeffrey

This paper details an efficient general method and a Maple implementation for the direct numerical evaluation of the Wright ω function to arbitrary precision over $\mathbb{C}$. Because ω is discontinuous along two rays in $\mathbb{C}$ this is nontrivial, and the implementation demonstrates the utility of computer algebra support for signed zero and control of rounding modes.

Categories and Subject Descriptors: G.1.2 [**Numerical Analysis**]: Special function approximation; G.1.5 [**Numerical Analysis**]: Roots of nonlinear equations—*convergence, error analysis, iterative methods*; G.4 [**Mathematical Software**]: Algorithm design, efficiency

General Terms: Arbitrary precision floating-point evaluation over $\mathbb{C}$

Additional Key Words and Phrases: Discontinuity, progressive precision

1. INTRODUCTION AND RATIONALE

The Wright ω function is defined in [Corless and Jeffrey 2002] to be

$$\omega(z) = W_{\mathcal{K}(z)}\left(e^z\right), \tag{1}$$

where $\mathcal{K}(z) = \lceil(\mathrm{Im}(z)-\pi)/(2\pi)\rceil$ is the *unwinding number*, which has the property that $z = \ln e^z + 2\pi i\mathcal{K}(z)$, and W_k is the kth branch of the Lambert W function, which satisfies $W_k(z)\exp W_k(z) = z$ for each k. See [Corless et al. 1996] for the properties of the Lambert W function. See [Corless and Jeffrey 2002] for properties of the Wright ω function, and [Corless and Jeffrey 2004] for discussion of how to implement support for this function in the computer algebra language Maple [Monagan et al. 2001]. In this paper we will show how to numerically evaluate ω to arbitrary precision efficiently in a computer algebra environment; again, without loss of generality, we will use Maple. We hope to follow this paper up with another describing robust and efficient evaluation of ω to hardware floating-point precision (double precision in languages supporting the IEEE standard [ANSI/IEEE 1985].

We will demonstrate in this paper that the direct numerical evaluation of this function is to be preferred over using the definition (1) via composition of numerical evaluation of W with numerical evaluation of exp. As a first example, the convex conjugate example of [Borwein and Lewis 2000], also described in [Corless and Jeffrey 2004], requires the computation of $\omega(x)$ for large x. Since $\omega(x) \sim x$ this should not cause overflow; but using Lambert W, one computes $W(e^x)$, which may

Ontario Research Centre for Computer Algebra
and the Department of Applied Mathematics
The University of Western Ontario
London, Ontario, Canada
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0098-3500/20YY/1200-0001 $5.00

well overflow on e^x. The main reason, however, is that the definition combines the *discontinuous* function $\mathcal{K}$ with W_k to form a function that is continuous in most places.

In this paper we give a direct method for the numerical evaluation of ω. The method is interesting because ω has two rays of discontinuity: $z = t \pm i\pi$ for $t \leq -1$, as described in [Corless and Jeffrey 2002]. Technically, it is impossible to evaluate ω accurately near these discontinuities, unless we can reliably determine whether or not $\mathrm{Im}(z) \geq \pi$, $\mathrm{Im}(z) < \pi$, $\mathrm{Im}(z) \geq -\pi$, or $\mathrm{Im}(z) < -\pi$. But this is the 'constant problem', which is not solved even for algebraic numbers unless Schanuel's conjecture is true [Richardson 1992]. This paper and the method implemented does not solve the problem: working over floats fixed at some arbitrary precision, equality with $\pm\pi$ is declared aggressively if the floating point representations are identical. In the presence of data or rounding errors this will lead to substantial mistakes—but this is inherent in the problem and cannot be avoided.

Even after a declaration has been made, say $\mathrm{Im}(z) \geq \pi$, then subsequent computations must be regularized to ensure compliance with this declaration. This is the main contribution of the method presented here, in that we give such a regularization, that may be used with any numerical method for the evaluation of ω, and may provide a model for the numerical solution of other logarithmic polynomials. In the case of ω, the solution over $\mathbb{C}$ of the logarithmic polynomial

$$y + \ln y = z$$

is, in the absence of a signed zero,

$$y = \begin{cases} \omega(z)\ , & z \neq t \pm i\pi, t \leq -1\ , \\ \omega(z), \omega(z - 2\pi i)\ , & z = t + i\pi, t \leq -1\ , \\ \text{no solution}\ , & z = t - i\pi, t \leq -1\ . \end{cases} \tag{2}$$

For a proof, see [Corless and Jeffrey 2002].

Another feature of the method described here is the analysis of a family of arbitrary-order iterative methods. For most equations $f(x) = 0$, higher-order iterative methods are not practical because of the difficulty of computing the higher derivatives of $f(x)$. For the present computation, however, the Taylor series of the appropriate functions can be written down and efficiently evaluated at an arbitrary regular point. Before the resulting schemes can be used, however, we have to analyze their numerical stability, and this analysis, as well as the family of methods, is new to this paper.

A direct numerical evaluation of ω over $\mathbb{C}$ is desirable for several reasons. First, it has already been pointed out that for large z, the expression $W(e^z)$ is difficult to evaluate. More importantly, there is a difficulty with spurious discontinuity. This is most easily demonstrated with an example computation in Maple: Numerically evaluating `omega( -0.9 + I*Pi )` by way of the definition (1) uncovers a subtle difficulty: When Maple evaluates `ceil( (Im(z)-Pi)/(2*Pi) )`, it uses some symbolic processing, and hence computes $\mathcal{K}(z) = 0$ correctly, by cancelling the symbolic `Pi`. But `exp(-0.9+I*Pi)` is left alone, until the user calls `evalf`. Then something unexpected happens: at Maple's default 10 decimal digits, `Pi` rounds to something larger than π; this then gives us a *negative imaginary part* on the order of roundoff in the result of the call to `exp`. This is all explainable in terms of the Maple model

of floating-point arithmetic, but it's a surprise nonetheless—one made visible by the next step, the computation of $W_0(x - i \cdot \varepsilon)$, which is *on the wrong side of the branch cut*. The numerical value of $W_0(x - i \cdot \varepsilon)$ is not at all close to the value of $W_0(x + i \cdot \varepsilon)$, and this discontinuity is *spurious*. The ω function is *continuous* at this point. This alone justifies having a separate routine for the numerical evaluation of ω, one that guarantees that we get continuity (where ω is continuous). The difficulty with the definition (1) is that it combines *discontinuous* functions in such a way that their discontinuities (mostly) cancel. We remark that this idea, mathematically combining discontinuous functions to make a continuous one, is done in other circumstances, such as production of continuous antiderivatives [Jeffrey 1994].

2. REVIEW OF RELATED WORK

For real values of the Lambert W function, accurate and efficient fixed-precision routines have been published [Fritsch et al. 1973; Barry et al. 1995]. Maple can evaluate W over the complex plane to arbitrary precision, as can Mathematica[1], but the algorithm has not been given in any detail [Corless et al. 1993; Corless et al. 1996]. Both [Fritsch et al. 1973] and [Barry et al. 1995] use the same rational fourth-order formula as a basic iteration method. The method in equation (7) below can be similarly cast as a rational function in r instead of a polynomial in r, potentially at some savings in evaluation cost. However, we will see in section 3.4 below that the dominant cost of the arbitrary-order methods is the generation of the series coefficients at each iteration, not the evaluation of the formula. Hence our implementation does not construct rational formulae, but contents itself with polynomials.

Another significant difference is that both [Fritsch et al. 1973] and [Barry et al. 1995] are concerned with at most double-precision evaluation, whereas we wish to have efficient formulae for arbitrary precision. The fact that both [Fritsch et al. 1973] and [Barry et al. 1995] study formulae for real values only is of lesser significance.

W. Kahan has implemented a variation of this same method in Matlab (W. Kahan, private communication). Relying in a clever way on features of IEEE arithmetic, the code is robust and very efficient, even vectorized, though again restricted to real values at the moment; however, he comments that it could be extended to arbitrary branches of W and arbitrary precision, with little effort.

The numerical evaluation of the Lambert W function in Maple is more comparable to what we are describing in this paper: it works to arbitrary precision over $\mathbb{C}$, using a third-order method (Halley's method) and progressive precision. It is very efficient, for an arbitrary precision method. For a description of this method, see [Corless et al. 1993; Corless et al. 1996].

We remark again, however, that we are describing a direct method to evaluate the Wright ω function, not the Lambert W function, in order to avoid problems with spurious overflow and discontinuities: this is not a paper about the evaluation of the Lambert W function.

[1] Mathematica calls the W function `ProductLog`.

3. ARBITRARY PRECISION EVALUATION OVER $\mathbb{C}$

We begin with a short discussion of the Maple model of arbitrary-precision floating point arithmetic. For details, see the help pages in Maple for `numerics`, `numeric_refs`, and `maple_floats`. In brief, Maple implements an extension of the IEEE 754/854 standard [ANSI/IEEE 1985; 1987], with a large family of available floating point numbers $F(B, p, L, U)$ with constants that may vary slightly on different platforms. On Windows XP machines, and on Intel Linux machines, the constants are: base $B = 10,000$, smallest exponent $L = 2-2^{31} = -2,147,483,646$, largest exponent $U = 2^{31} - 2 = 2,147,483,646$, and any precision p in $1 \le p \le 2^{27} - 8 = 268,435,448$. Any floating point number f at a given precision p is represented by a sign s, a normalized significand d of p decimal digits, and an exponent e satisfying $L \le e \le U$. We have $f = s \times 0.d_1 d_2 \ldots d_p \times 10^e$. Basic operations are guaranteed to give the infinitely precise results exactly rounded to the nearest floating point number representable in the system with p digits. The exponent ranges are large in comparison to those of arithmetics supported by most compiled languages, though other extended-range arithmetics exist (see for example [Lozier and Smith 1981; Smith et al. 1981; Schonfelder 2001]).

Even though Maple's floating-point system conforms (quite stringently) to the IEEE standard, where it applies, it is different from other multiple-precision packages, such as either the Fortran package of R. P. Brent [Brent 1978] or David Bailey's MPFUN package [Bailey 1993], in several respects. One major difference is that Maple is an interpreted language, and hence *the relative cost of changing the precision in the middle of a computation is negligible*, in comparison with the overhead of interpretation. Maple simply switches between virtual machines at different precisions p; if the new precision p_2 is higher than the old precision p_1, no operation needs to be done on the input, but if $p_2 < p_1$ then the input must be rounded to p_2 digits. Experiments show that indeed this operation is of negligible cost. This will allow us to use *progressive precision* in our iterative scheme: that is, we may vary the precision as needed, increasing it at each iteration as the computation proceeds, and decreasing it on a fine grain during the generation of the iteration formula for the next step. We do not claim for Maple an *absolutely* efficient technique for switching precision, but merely note acceptance (under certain conditions, namely in comparison with exact arithmetic and its even greater cost) of fairly slow floating-point arithmetic in general; under these conditions, changing precision does not impose a noticeable further penalty. However, the slow, as we term them here, arbitrary-precision floating-point environments of Maple and of other computer algebra languages (see e.g. [Fateman 1976]) have proved both popular and useful, even with the compromise of some speed in return for convenience.

Maple uses a polyalgorithm for the multiplication of floating-point numbers. At low precisions p Maple uses the classical $O(p^2)$ algorithm; for moderate precisions, Maple uses Karatsuba multiplication, which has complexity $O(p^\lambda)$ where $\lambda = \ln 3/\ln 2 \approx 1.585$. For the largest precisions Maple could use an FFT multiplication, with complexity $O(p \log p)$, but at present does not, in part because for truly large precisions it would be better to use a special-purpose package such as that of [Bailey 1993].

As of Maple 9, Maple uses Gnu MP arbitrary-precision library for integer arith-

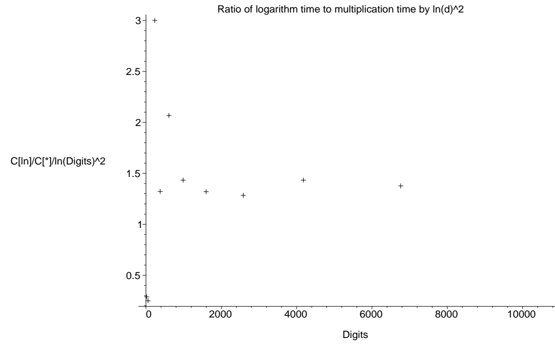

Fig. 1. The ratio of the cpu time of taking the log of a p-digit number in Maple to the cpu time of multiplying two p digit numbers together in Maple, divided by $\ln^2 p$.

metic. Plans are being considered to move to Gnu MP [Granlund 2004] arbitrary-precision floating point arithmetic, which in addition to using the FFT for large enough precisions, also may use Toom-Cook 3-way multiplication, with asymptotic cost $O(p^\mu)$ with $\mu = \ln 5/\ln 3 \approx 1.465$ as an intermediate algorithm.

Maple uses an AGM-like algorithm to compute the exponential function; to compute the logarithm it inverts $x = \exp y$ using an iterative scheme (there is no published description of the details of Maple's method, aside from the publicly disclosed portions of source code, at this time of writing). Experimentally, we find (see Figure 1) that the cost of computing logarithm is approximately $1.5 \ln^2 pM(p)$, over a wide range of p, where $M(p)$ is the cost to multiply two floating-point numbers at precision p. This is not optimal for large p in that algorithms exist that are $O(M(p) \ln p)$ (see [Brent 1976]), but represents a practical choice for moderate-sized p (Gaston Gonnet, private communication). Also, the exponential function is implemented in Maple in compiled kernel code, improving the speed of evaluation; in contrast the logarithm is implemented in interpreted library code.

One consequence of this is that our present algorithm, which solves $y + s \ln y = z$ to evaluate $\omega(z)$, relies on a somewhat slower logarithm than it could; moreover, the solution of $u \exp u = x$ for the Lambert W function relies on the inversion of the faster (as implemented) exponential function. Therefore we expect our Wright ω algorithm to be slower than that for Lambert W; however, for the reasons already discussed, we believe that the improved accuracy in the face of discontinuities and the avoidance of spurious discontinuities and spurious overflow more than justify the extra computation time. Future versions of Maple could alter these speed considerations.

3.1 An arbitrary-order iterative method

The basic method we use is an Nth-order variant of Newton's method (see e.g. [Corless et al. 1996] or the references therein), to compute ω as the root of $y + \ln y = z$, when z is not near the discontinuities of Wright ω. For this function (and indeed for any function for which we can easily compute its Taylor series at any point) we may form Nth order methods by truncation of the arbitrary-point series and

iteration of the resulting formula [Corless et al. 1997]. The Taylor series for ω about $z = a$ is as follows. Let $\omega_a = \omega(a)$; then

$$\omega(z) = \sum_{n\geq 0} \frac{q_n(\omega_a)}{(1+\omega_a)^{2n-1}} \frac{(z-a)^n}{n!} \tag{3}$$

where

$$q_n(w) = \sum_{k=0}^{n-1} \left\langle\!\!\left\langle {n-1 \atop k} \right\rangle\!\!\right\rangle (-1)^k w^{k+1} , \tag{4}$$

where $\left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle$ is a second-order Eulerian number [Graham et al. 1994]. The series (3) will be used heavily in what follows.

Here are iteration formulas $w_{k+1} = F_N(w_k)$ of various orders N, as derived by truncating the series (3). In each equation, the residual $r = z - w - \ln w$.

$$F_2(w) = w + \frac{w}{1+w} r \tag{5}$$

$$F_3(w) = w + \frac{w}{1+w} r + \frac{w}{2(1+w)^3} r^2 \tag{6}$$

$$F_4(w) = F_3(w) - \frac{w(2w-1)}{6(1+w)^3} r^3 \tag{7}$$

In short, and scaling with $\rho = r/(1+v)^2$, where $v = sw$ with $s = \pm 1$ chosen later for regularization, we have

$$F_N(w) = w + (1+v) \left(w\rho + \sum_{j=2}^{N-1} C_j \rho^j \right) . \tag{8}$$

The first two terms of the error expansion are

$$\omega(z) - F_N(w) = (1+v) \left(C_N \rho^N + C_{N+1} \rho^{N+1} \right) + O(\rho^{N+2}) \tag{9}$$

We have seen by direct computation that the zeros, aside from $w = 0$, of all the $C_j = q_j(w)/j!$ that we need for this computation are all real, positive, and *interlace*,[2] and so there is no value of $w \neq 0$ for which more than one C_j is zero. Therefore taking two terms in the error formula suffices for a valid estimate for a stopping criterion, asymptotically as $w \to \omega(z)$.

We may generate the C_j by the following recurrence relation, easily derived from the equation $y + \ln y = z$ satisfied by $\omega(z)$ away from the branch points, and even on the branch cuts if care is taken by regularization:

$$C_{k+1} = \frac{1}{k+1} \left((1+v)C_k - s \sum_{j=1}^{k} (k+1-j) C_{k+1-j} C_j \right) . \tag{10}$$

[2]We conjecture that this is true for all j. We have not attempted a proof. We note that the low-order terms of the polynomials are $1 - (2^n - 2n)w + O(w^2)$ and hence the smallest root approaches $1/(2^n - 2n)$ as $n \to \infty$; the largest root seems to tend to $2n$; undoubtedly there is an explanation of this in terms of 2nd order Eulerian numbers, but we do not pursue this further at this time.

Table I. The residuals during the iteration of equation (7) and of an eleventh-order iteration obtained by truncating the series (3) keeping the r^{10} term. The starting value was $w_0 = 0.8$, and residuals were computed in sufficiently high precision.

n	$r_n^{(4)}$	$r_n^{(11)}$
0	$-0.18856 \cdot 10^{-1}$	$-0.1886 \cdot 10^{-1}$
1	$0.244 \cdot 10^{-9}$	$-0.859 \cdot 10^{-26}$
2	$-0.708 \cdot 10^{-41}$	$-0.150 \cdot 10^{-293}$
3	$-0.499 \cdot 10^{-167}$	$-0.672 \cdot 10^{-3239}$
4	$-0.123 \cdot 10^{-671}$	$-1.0 \cdot 10^{-35638}$

We have $C_0 = w/(1+v)$. We study the numerical stability of this recurrence relation in section 3.3 below.

Alternatively, the polynomials $q_j(w)$ may be computed from the following recurrence relation [Corless et al. 1996]:

$$q_{n+1}(w) = -(2n-1)wq_n(w) + w(w+1)q_n'(w) , \tag{11}$$

where $q_1(w) = w$.

Table I shows the residuals in a fourth order iteration and in an eleventh order iteration for $\omega(0.558)$. As one can see, it takes four iterations of the fourth-order method to get to 600 digits of accuracy, but only three iterations of the eleventh-order method, which in fact may get over 3000 digits accuracy; in practice, two iterations of the eleventh order method plus one iteration of the second order method gets 600 digits correct for this example. The cost of each iteration method is of course different, but both cases is dominated by the cost of evaluating the logarithm in the residual $r = z - w - \ln w$.

3.1.1 *Choosing the order for each iteration.* The method must compute at least one residual at the full desired precision, because the iterative formulas all require an accurate residual. Therefore the most effective optimization that can b done is to make the cost of the penultimate residual computations as small as possible, while not making the cost of the ultimate generation and evaluation of formulae too large. High-order formulae allow the previous residual to be imprecise, but are expensive to generate and evaluate. By choosing the highest order whose cost of generation and evaluation is no more expensive than the cost of evaluating the residual, we believe that we could minimize the overall cost of the final two steps, which dominate the cost of the iteration.

Another possible strategy is to take as high an order a formula as possible, with at most the same cost as computation of the residual, on each iteration, until the final iteration, when a lower-order formula (equivalent to Halley's method or Newton's method, usually) suffices to bring the answer to the desired precision. This is the strategy implemented in the code at present; we plan further experiments to test whether or not the strategy of the previous paragraph is better in practice than this one, which may be better because it takes best advantage of progressive precision by using a low-order formula for the final iteration (see Section 3.5).

3.2 Iteration near the branch points

Iterative schemes of the type given in section 3 are known to drop to first-order convergence at repeated roots. At the branch points $z = -1 \pm i\pi$, where $\omega = -1$, the schemes (5)–(7) clearly break down. However, because we know the value of ω at the branch points, we need only iterate *near* these points. Once the estimate for ω is sufficiently close to its final value, the formulae return to their stated convergence rate. We here show that if the residuals r_k are scaled by Δ^2 where $\omega(z) = -1 + \Delta$, that is $r_k = \rho_k \Delta^2$, then the iterative schemes here give $\rho_{k+1} \sim C_N \rho_k^N$ for a constant C_N depending on the method, and hence the convergence attains the stated order if ρ_0 is smaller than 1. Good starting guesses are therefore essential for good performance.

The key to proving this is the observation that, with $\omega_k = \omega(z - r_k)$,

$$\omega(z) = \omega(z - r_k + r_k) = \omega_k + \frac{\omega_k}{1+\omega_k} r_k + \sum_{j=2}^{\infty} \frac{q_j(\omega_k)}{j!(1+\omega_k)^{2j-1}} r_k^j$$

or

$$\omega(z) = \omega_k + (1+\omega_k)\left(\omega_k \rho_k + \sum_{j=2}^{N-1} \frac{q_j(\omega_k)}{j!} \rho_k^j\right) + (1+\omega_k) C_N \rho_k^N + \cdots$$

where $C_N = q_N(\omega_k)/N!$. The residual r_{k+1} is related to the forward error $\omega_{k+1} - \omega_k$ by the conditioning of the problem:

$$\omega_{k+1} - \omega_k \approx \frac{\omega_k}{1+\omega_k} r_{k+1}$$

and, since the forward error is asymptotically the first term neglected in the series above, we have that

$$r_{k+1} = \frac{(1+\omega_k)^2}{\omega_k} C_N \rho_k^N + \cdots$$

or, dividing both sides by $(1+\omega_k)^2$ and using the fact that $\omega_{k+1} = \omega_k + O(\rho_k)$, we have

$$\rho_{k+1} = \frac{C_N}{\omega_k} \rho_k^N + \cdots .$$

If $C_N(\omega_k) = 0$ by happenstance, then the next residual is of even higher order. In the scaling used in the first paragraph, note that $\omega_k + 1$ tends to Δ. One can also show that this series converges if $|\rho| \le 1/2 - O(\Delta)$ for $\Delta \to 0$. This highlights the importance of a good initial guess, whence we need $|\rho_0| < 1/2$.

3.3 Numerical stability of the iteration formulae

We have two concerns about the iteration formulae. The first is their numerical stability, or lack thereof, for evaluation purposes. The second is the numerical stability, or lack thereof, of the process used to generate them.

3.3.1 *Condition of evaluation.* At each iteration we compute

$$w_{k+1} = F_N(w_k) = w_k + (v_k + 1) \sum_{j=1}^{N} C_j(w_k) \rho_k^j \tag{12}$$

where $\rho_k = r_k/(1+v_k)^2$ and $r_k = z - v_k - \ln w_k$. We use Horner's method to evaluate this polynomial, which is reasonably stable (i.e., equivalent to exact evaluation of a nearby polynomial), but we need to know if this polynomial in ρ is ill-conditioned for evaluation (i.e. sensitive to data errors or to changes in its coefficients. As a polynomial in ρ, its condition number with respect to *relative* perturbations or errors in the coefficients C_j is given by (see e.g. [Farouki and Rajan 1987])

$$C = \sum_{j=1}^{N} |(v_k+1)C_j(w_k)\rho^k| \,. \tag{13}$$

To ensure numerical stability of evaluation in this variable precision environment, we need only add $\lceil \log_{10} C \rceil$ guard digits during the evaluation. By computing a hardware-floating point estimate of C with our initial estimate w_0 we may do this at little cost.

3.3.2 *Condition of generation.* A short perturbation computation shows that, if each C_j is perturbed to $C_j + \delta_j$ for $0 \le j \le k$, then the resulting perturbation of C_{k+1} is, to leading order, polynomial in the initial value $\omega(z - r) = w = w_0$, and of course linear in each δ_j. However, a full analysis along these lines is quite complicated, because since the recurrence relation is nonlinear, initial errors and rounding errors interact in nonlinear ways. We present a simplified analysis below that assumes only a single initial error $C_0 \mapsto C_0 + \delta_0$ and studies its propagation through a linearized recurrence relation. Let

$$\hat{C}_k = \frac{1}{k}\left(\hat{C}_{k-1}(1 + w_0 + \delta_0) - \sum_{j=1}^{k-1}(k-j)\hat{C}_j\hat{C}_{k-j}\right) \tag{14}$$

and $\hat{C}_k = C_k + \delta_k$ for $k \ge 1$. Brute computation with Maple shows that, ignoring higher-order terms, the first five coefficients are perturbed by

$$\begin{aligned}
\delta_1 &= \frac{\delta_0\,(2\,w_0+1)}{1+w_0}\\
\delta_2 &= -1/2\,\frac{\delta_0\left(5\,{w_0}^2-2\,w_0-2\right)}{1+w_0}\\
\delta_3 &= 1/6\,\frac{\delta_0\left(25\,{w_0}^3-31\,{w_0}^2-11\,w_0+4\right)}{1+w_0}\\
\delta_4 &= -1/24\,\frac{\delta_0\left(175\,{w_0}^4-357\,{w_0}^3+10\,{w_0}^2+81\,w_0-8\right)}{1+w_0}\\
\delta_5 &= \frac{1}{120}\,\frac{\delta_0\left(1575\,{w_0}^5-4377\,{w_0}^4+1702\,{w_0}^3+1022\,{w_0}^2-363\,w_0+16\right)}{1+w_0}
\end{aligned}$$

This is enough experimentation to see the general idea. We see that, since some of the C_k may be zero for particular values of w_0, the relative error introduced is potentially infinite, even if the absolute error is small. However, we are not interested in just computing these coefficients C_k: what we want is their sum, multiplied by $(1+w_0)$ and ρ^k. When a coefficient C_k is nearly zero and so the relative error is large, the error in the sum itself is not large because we will still

be adding a small term, using again the fact that the zeros of the q_j interlace, and hence not all terms may be accidentally zero. That is, it is the absolute error that matters here.

We also see that for large w_0 that the perturbation in δ_k behaves roughly as $O(\sigma_k w_0^k)$; by inspection, we see that the ratios σ_{k+1}/σ_k tend to -2, apparently (it is -1.95 at $k=19$, $-1.9666\ldots$ at $k=29$, and -1.975 at $k=39$; these are $-2+1/20$, $-2+1/30$, and $-2+1/40$, respectively). Again, of course, these coefficients are multiplied by $(1+w_0)$ and by ρ^k, and so the growth in the error is attenuated in the sum if $|-2\rho w_0| < 1$.

It is also of interest to examine what happens when $w_0 \approx -1$. We see that at $w_0 = -1$, these constants are

$$[-1, -5/2, -\frac{41}{6}, -\frac{151}{8}, -\frac{6253}{120}, \ldots]$$

and the ratios of these numbers tend to a number less than 2.682 (as indicated by a graph of the first 41 such ratios), but apparently larger than 2.68. Thus we see that initial errors of order δ_0 are amplified by a factor approximately $O(2.68^k)$, for w_0 near to -1. This is surely an unstable recurrence relation.

However, as before, these errors are attenuated in the sum whenever $2.682|\rho| < 1$; remember also that for convergence of the sum (which we do not need because we take only a finite number of terms, of course) $2|\rho| < 1$ is necessary; for numerical stability of the sum it seems that we need only be slightly more accurate with our initial guess than we would have to be if we were using an infinite number of terms and exact arithmetic.

We stress that the above analysis is only partial: we have not shown (and do not know) whether or not floating-point evaluation of the recurrence relation is equivalent to making a single error in the initial value.

3.4 Complexity

The theoretical complexity of the solution of nonlinear equations containing logarithm, such as the equation for computing ω, is well-known to be asymptotically just the cost of evaluating logarithm, $O(M(n)\ln n)$, where $M(n)$ is the cost to multiply two n-digit numbers [Brent 1976; van der Hoeven 1999]. We have observed experimentally that the cost of our code is asymptotically 2.7 times the cost to compute logarithm. See Table II.

Table II. Some representative timings; $z = 0.5585 + 0.332i$

Digits	time $\omega(z)$ (sec)	time $\ln(1+z)$ (sec)
2000	13	2.25
4000	28	7.6
20000	122	44.8
100000	493	178

3.5 The effect of progressive precision

Many algorithms in Maple, and perhaps in other computer algebra languages, start their iterations at low precision and increase precision as the iterations proceed. We refer to this process as *progressive increase of precision* during the iteration.

Progressive precision can also be used on a finer scale, for the generation of the iteration formula using equation (10) where, given N to start with, the precision can be decreased as successive C_j's are generated, and the precision can be progressively increased again during a Horner's method evaluation of the sum.

During series generation, we decrease precision by a constant amount $p/(N+1)$, starting from precision p plus some guard digits g to compensate for any instability in series generation or evaluation. We assume $g \ll p$. We also model the cost of multiplication at precision p to be $C_* p^\mu$, where $\mu = \ln 3/\ln 2 \approx 1.585$ corresponds to Karatsuba multiplication. Therefore, the cost to compute C_k for $0 \le k \le N$ may be modelled as $\sum_{k=1}^{N} \mathtt{Cost}(C_k)$, where

$$\mathtt{Cost}(C_k) = C_* \left(2(k-1)+1\right)\left(p+g-kp/(n+1)\right)^\mu . \tag{15}$$

We model the cost of a division as approximately equal to one multiplication (experimentally this seems to be true in Maple). By replacing the sum with an integral, we find that the dominant asymptotics of the sum is

$$\mathtt{Total\ Cost} \approx 0.22\, N(N+1.8)\left(\frac{N}{N+1}\, p\right)^\mu C_* \tag{16}$$

Without progressive precision, generation of the C_k would cost $C_* N(N+1)p^\mu$, in comparison. Thus, asymptotically for large orders N, progressive precision reduces the cost to about $1/5$ the cost it would be without it. For small N, of course, direct comparison is a better model.

During the evaluation of the iteration formula by Horner's method, we increase precision on a fine-grain scale by the same constant amount $p/(N+1)$ at each iteration of the Horner loop. This implies that the cost of evaluation of the formula is, to leading order,

$$N C_* p^\mu \left(\frac{\ln 2}{\ln 2 + \ln 3}\left(\frac{N}{N+1}\right)^\mu\right) \tag{17}$$

compared with $N C_* p^\mu$ for nonprogressive evaluation. Asymptotically for large N we see that this gains a factor of approximately $\ln 2/(\ln 2 + \ln 3) \approx 0.387$. So, again, progressive precision saves a constant factor, in this case about 60%, of the cost of evaluation of the formula for large precisions. However, this cost is only linear in N.

We see that the cost of generating the series dominates, for large N, being quadratic; except of course for the cost of the evaluation of the residual, which involves the computation of a logarithm. As shown in Figure 1, that is approximately $1.5 \ln^2 p$ times the cost of a multiplication, which is proportional to p^μ. Therefore, the cost of the p-digit logarithm will dominate for all orders up to N such that $0.22N(N+1.8) = 1.5\ln^2 p$, or (for large p), $N \approx 2.7 \ln p \approx 6 \log_{10} p$. For $p = 100$ digits, this is about $N = 12$, whereas for $p = 10{,}000$ this is $N = 24$.

Therefore, a "greedy" strategy of using as high an order of iteration as is available (up to $2.7 \ln p$) to get to the desired precision in as few iterations as possible may

save on the total cost by saving on the number of evaluations of logarithm (once per iteration). Typically the final step will be taken with a lower-order method because, say, the $(n-1)$st iterate will be accurate to $D/2$ or $D/3$ digits, and in that case only a 2nd or 3rd order method is needed to get to D digit accuracy, and there is an even further gain from fine-scaled progressive precision on that last iteration.

Because residuals are computed every iteration except the last (see section 3.8), and because we have $w_k = \omega(z - r_k)$ exactly, we know at each stage how accurate our iteration is, and can predict roughly how accurate our next iteration will be. By computing at low precision the coefficients of the Taylor series, we can predict how accurate our next iteration will be with any order formula, and hence we may choose the formula of least cost for the final step.

In practice, we restrict to orders from $N = 1$ (corresponding to Newton's method) to $N = \texttt{Nmax} = 20$. The cost to evaluate the iteration formulae grows with N, as we have shown, but also the numerical instability of the iteration formula grows with N as well, as pointed out in section 3.3.

3.6 Starting with hardware precision

Every iterative scheme needs a starting guess. This package uses as a starting guess the results of a "hardware floating-point" function for the evaluation of ω over $\mathbb{C}$. What is meant here by "hardware floating-point" is a routine written in entirely real arithmetic, suitable for use with Maple's `evalhf` facility, that accepts (x, y) as input (plus some other information that may be gleaned at higher precision, such as whether or not $y \geq \pi$ to the input precision) and returns a "hardware precision" value of $\omega(x + iy)$; in our current version, the hardware floating point function is accurate to 13 decimal digits (in the residual), and this suffices to start the iteration described in this paper.

The method used by the routine will be described in a future paper, but in brief it uses two iterations of the Fritsch-Schafer-Crowley formula [Fritsch et al. 1973], adapted for the regularized iteration of $sw + \ln w = \zeta$, from a starting guess that is derived from the series at either branch point, at $-\infty$, the asymptotic series for the outsides of the doubling line and its reflection, the Taylor polynomial at $z = 1$, and the asymptotic series at ∞.

3.7 Convergence

For a fixed z, we use the iteration formula $w_{k+1} = F_N(w_k)$.

Theorem. If $|F'(w)| \leq \tilde{\rho} < 1/\sqrt{2}$ for all $\omega \in C \subset \mathbb{C}$, then C is a contracting region for F, and there exists a unique $\omega^* \in \mathbb{C}$ such that $\omega^* = F(\omega^*)$ and $\omega_k \to \omega^*$ as $k \to \infty$. Moreover, $|\omega_{k+1} - \omega^*| \leq \sqrt{\rho_x^2 + \rho_y^2}\,|\omega_k - \omega^*|$ where $\sqrt{\rho_x^2 + \rho_y^2} \leq 1$.

Proof. The main statement is a simple consequence of the general fixed-point theorem. For the second statement, we note that $\omega_{k+1} - \omega^* = F(\omega_k) - F(\omega^*)$, and, by the complex mean-value theorem [Evard and Jafari 1992], there exist ω_1 and ω_2 lying on the line segment joining ω^* and ω_k such that

$$\mathrm{Re}\,(F'(\omega_1)) = \mathrm{Re}\left(\frac{F(\omega_k) - F(\omega^*)}{\omega_k - \omega^*}\right) = \rho_x \tag{18}$$

$$\mathrm{Im}\left(F'(\omega_2)\right) = \mathrm{Im}\left(\frac{F(\omega_k) - F(\omega^*)}{\omega_k - \omega^*}\right) = \rho_y \tag{19}$$

Thus $[F(\omega_k) - F(\omega^*)]/[\omega_k - \omega^*] = \rho_x + i\rho_y$, or

$$\omega_{k+1} - \omega^* = (\rho_x + i\rho_y)(\omega_k - \omega^*) \ . \tag{20}$$

Whence $|\omega_{k+1} - \omega^*| \le \sqrt{\rho_x^2 + \rho_y^2}\,|\omega_k - \omega^*|$, and by hypothesis, $\sqrt{\rho_x^2 + \rho_y^2} \le 2\tilde{\rho} < 1$. □

Theorem. Given an initial estimate $\omega_0(z)$, if $|F'| < 1/\sqrt{2}$ at $\omega = \omega_0(z)$, then the iteration will converge.

Proof. If $|F'(\omega)| < 1\sqrt{2}$, then $\omega_0(z) \in C$ for some contracting region C. Since away from the branch cut, the solution is unique, we must converge to the correct root.

One method to test that the starting guesses are in the contracting region is to plot $|F_N'(w_0(z))|$ for all z. We have done this[3], and all derivatives are less than 0.01 and hence less, *a fortiori* less than $1/\sqrt{2}$.

3.8 Conditioning, and tests for convergence

A residual for $y + \ln y = z$ is computed at every step of the iteration, giving $\omega_k + \ln \omega_k - z = -r_k$, which can be interpreted as follows: the computed ω_k is the exact value of $\omega(z - r_k)$. Wherever ω is continuous, $r_k \to 0$ implies convergence of ω_k to $\omega(z)$. More precisely, we may compute the condition number of ω near z by the derivative formula $\omega'(z) = \omega(z)/(1 + \omega(z))$, and thus get an asymptotically correct estimate of the forward error from the residual:

$$\omega(z) - \omega_k \approx \frac{-z\omega_k}{1 + \omega_k}\frac{r_k}{z} \ . \tag{21}$$

We terminate iteration when this quantity is smaller than the error requested by the user. We also terminate when the relative residual r_k/z is smaller than can be distinguished (which is variable with the setting of Digits). Near $z = 0$, we use absolute error for the residual instead of the relative error above.

3.9 Signed zero, rounding modes, guard digits, and input precision

Maple has a signed zero, in accordance with the IEEE specification [ANSI/IEEE 1985]. In fact, it has six signed floating-point zeros: $+0$, -0, and $\pm 0 \pm 0i$. It also has an exact zero, which behaves in some circumstances like the real $+0$. In Maple, the signed floating-point zeros are denoted, e.g., `-0.+0.*I`. The code described in this paper respects these signed zeros.

Maple allows control over rounding modes, in accordance with the IEEE specification. One sets the environment variable `Rounding` (in this case, to `-infinity`) to control the direction. Here, we need to set the rounding mode to ensure that the branch cuts are closed from below, which is different from that of many other elementary functions but comes directly out of the definition of Wright ω via the unwinding number, which itself is closed in its range from below.

[3]Of course, this means that all we did was sample the derivative at a number of points and looked at the resulting picture. We did not use the "honest plotting" of e.g. [Fateman 1992; Tupper 2001], and it is true that we may have been misled. But we do not believe so.

Guard digits are used in ill-conditioned regions; in particular, near the branch points, twice as many digits are used (because the branch points are second order). For large w_k, an additional $\lceil \log_{10} 2.68|w_k| \rceil$ guard digits are added to ensure that the series coefficients C_j for the iteration formula are generated accurately. Finally, $\lceil \log_{10} C \rceil$ guard digits are added, where C is the condition number of evaluation of the iteration formula $F_N(w_k)$, to ensure that w_{k+1} is computed accurately.

The final results are rounded to the input precision.

3.10 Regularization near the discontinuities

Computing ω close to the discontinuity lines has several dangers. Because floating-point approximations to π are sometimes larger than π and sometimes smaller, depending upon the precision used, there is a danger that an iterative calculation will switch from numbers slightly less than π to numbers slightly greater than π during a calculation. For high-precision calculations, it is common to work at different precisions during a calculation for efficiency. Since ω is discontinuous at $z = x + i\pi$, for $x < -1$, crossing from one side of π to the other can result in an iteration converging to the wrong value, or not converging. This happens, for example, at $z = -2.94 + i(\pi + 0.0000001)$, at 14 decimal digit precision, with a 6 term asymptotic series initial guess; the initial guess has the wrong sign for the imaginary part, and the iteration never recovers. To avoid this problem, we regularize the calculation.

If z is near the top branch cut (within 0.001), we carefully—at the input precision—subtract $i\pi$ from z. By 'careful', we mean that since the cut is closed from below, we set the rounding mode to round towards $-\infty$. If z is near the bottom branch cut, we add $i\pi$ in the same way. Recall that both lines are closed from below. We now transform the discontinuous problem $w + \ln w = z$ to a continuous one by putting $w = -v$ and solving

$$-v + \ln v = z - i\pi \tag{22}$$

or

$$-v + \ln v = z + i\pi \; . \tag{23}$$

as appropriate. If z is above the top branch cut, $w = W_1(\exp(z))$ has values in the range of W_1, which lies above the negative w-axis, between $-\infty < w < -1$. This means that $v = -w$ lies below the positive v-axis between $1 < v < \infty$. If z is below the bottom branch cut, $w = W_{-1}(\exp(z))$ has values in the range of W_{-1}, which lies below the negative w-axis, between $-\infty < w < -1$. Therefore $v = -w$ lies above the positive v-axis between $1 < v < \infty$. Moreover, since $W_1(x+0i) = W_{-1}(x-0i)$, v is continuous across this axis. Similarly, if z is below the top branch cut or above the bottom branch cut, v lies in the negative of the range of W_0 and is obviously continuous across $0 < v < 1$.

The branch cut for $w = \omega(z)$ is on the negative w-axis, which coincides with the branch cut for ln. Negating, i.e. taking $v = -w$, gives us the image of the regularized plane; note in this case that the branch cut for $-v + \ln v$ lies on the negative v axis, while the values of v we are interested in lie on the positive v axis. Moreover, the initial guess we provide is sufficiently accurate to deal with the fact that the mappings $\zeta = z - i\pi$ and $\zeta = z + i\pi$ introduce a potential double-valuedness.

Thus, Nth order iteration on $-v + \ln v = \tau$, where $\tau = z - i\pi$ if z is near the top cut, and $\tau = z + i\pi$ if z is near the bottom cut, will converge to the correct answer with no danger of branch switching. Moreover, the initial guesses are good enough to separate the cases $v < 1$ and $v > 1$, which regularizes the discontinuity. The fact that this works was verified experimentally by (accidentally) giving very bad initial guesses to the iteration, which converged anyway (albeit initially slowly).

As can be verified by computing the series solution to $-v + \ln v = \tau$ about $\tau = a$, the Nth order iteration formulas are the same for this regularized problem as they are for the original problem, apart from some sign differences. We use one routine to compute solutions to both problems, given a $+1$ or a -1 to distinguish the cases.

Care must be taken before computation begins to add or subtract π at the input precision, because, for example, if the input is `t+I*Pi` for some $t < -1$, then the user will expect the input to be exactly on the top cut; but floating-point evaluation of `Pi` will either be above or below the branch cut, depending on the rounding. This masks an intractable problem, but the routine aggressively declares that input that has imaginary part that evaluates at the user's input precision to what `Pi` evaluates to, exactly, is exactly on the branch cut. This is, in practice, difficult to implement for precision less than or equal to the hardware-floating point cutoff, because of binary-to-decimal conversion issues. Our implementation does not worry about this issue for such low-precision calculations.

Finally, if the result has imaginary part that is smaller than the input tolerance, then the information about where the input z was (above the doubling line, below or on it, above the reflection, below or on it) is used to determine the sign of the zero imaginary part that is returned. We must set the imaginary part to zero as it may have the wrong sign because of rounding or truncation errors in the iteration. Briefly, if z is near the doubling line we return $+0i$, but if it is near the reflection then we return $-0i$. This ensures correct closure, and also has the surprising effect of providing a valid numerical solution to $y + \ln y = z$ on the reflection $z = t - i\pi$ for $t \leq -1$, namely $W_{-1}(-\exp t) - 0i$. This does not contradict the theorem of [Corless and Jeffrey 2002] that says there is no solution to this equation on that line, because by including a signed zero we have extended the definition of the domain of solution.

3.11 Testing

These iterative methods are self-checking and self-correcting. The function was evaluated on and near the branch cuts, near $-\infty$ and complex ∞, near the regions where the initial guess functions change over, near 0, and at random points. Further testing is always welcome, and the code (a prototype version not yet incorporated into Maple) is available for inspection.

3.12 Performance

The method is very fast. The routine, implemented in the interpreted language Maple, has a cpu time cost to compute one evaluation of $\omega(z)$ that is asymptotically 2.7 times the cost to evaluate one logarithm at the requested precision, for all settings of Digits tested between 100 and 100,000 Digits.

4. CONCLUDING REMARKS

A package has been written in Maple to support the Wright ω function. It includes a robust and efficient method for numerical evaluation to arbitrary precision over $\mathbb{C}$.

A separate routine to evaluate the Wright ω function only to double precision over $\mathbb{C}$, but very quickly, would also be useful; indeed such a routine could be used to replace the initial guesses discussed in section 3.6.

The discussion of numerics in this paper concentrated on the difficult issues of how to compute near the discontinuities. These results and methods have intrinsic mathematical interest, and are described here in some detail for the first time. The results are interesting to members of the mathematical software community principally because the discontinuities (along the branch cuts) are especially visible, and nontrivial, in this function. Therefore the function makes a good test case for implementing complex-valued expressions in computer algebra systems.

Acknowledgements

Many people were very helpful with discussions during the writing of this paper and its accompanying code, especially Dave Hare, Edgardo Cheb-Terrab, and Jürgen Gerhard. Some testing of the efficiency of higher order codes was carried out by Hui Ding. Jon Borwein and Bill Gosper provided motivation to look at ω.

REFERENCES

ANSI/IEEE. 1985. *Standard for Binary Floating-Point Arithmetic: Standard 754-1985.* IEEE Press. Reprinted in ACM SIGPLAN Not. 22, 2 (1987), pp. 9-25.

ANSI/IEEE. 1987. *Standard for radix-independent floating-point arithmetic: Standard 854-1987.* IEEE Press, Piscataway, NJ.

Bailey, D. H. 1993. Algorithm 719; multiprecision translation and execution of fortran programs. *ACM Trans. Math. Softw. 19,* 3, 288–319.

Barry, D. A., Barry, S. J., and Culligan-Hensley, P. J. 1995. Algorithm 743; WAPR: a Fortran routine for calculating real values of the W-function. *ACM Transactions on Mathematical Software (TOMS) 21,* 2, 172–181.

Borwein, J. M. and Lewis, A. S. 2000. *Convex Analysis and Nonlinear Optimization.* Advanced Books in Mathematics. Canadian Mathematical Society.

Brent, R. P. 1976. Fast multiple-precision evaluation of elementary functions. *J. ACM 23,* 2, 242–251.

Brent, R. P. 1978. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4,* 1, 57–70.

Corless, R. M., Gonnet, G. H., Hare, D. E. G., and Jeffrey, D. J. 1993. Lambert's W function in maple. *Maple Technical Newsletter 9*, 12–22.

Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J., and Knuth, D. E. 1996. On the Lambert W function. *Advances in Computational Mathematics 5*, 329–359.

Corless, R. M. and Jeffrey, D. J. 2002. *On the Wright ω function.* LNAI, vol. 2385. Springer, Marseille, 76–89.

Corless, R. M. and Jeffrey, D. J. 2004. Computer algebra support for the Wright ω function. *submitted*.

Corless, R. M., Jeffrey, D. J., and Knuth, D. E. 1997. A sequence of series for the Lambert W function. In *ACM International Symposium on Symbolic and Algebraic Computation.* 195–203.

Evard, J.-C. and Jafari, F. 1992. A complex Rolle's theorem. *American Mathematical Monthly 99,* 9, 858–861.

Farouki, R. T. and Rajan, V. T. 1987. On the numerical condition of polynomials in Bernstein form. *Comput. Aided Geom. Des. 4,* 3, 191–216.

Fateman, R. 1992. Honest plotting, global extrema, and interval arithmetic. In *ACM International Symposium on Symbolic and Algebraic Computation*. ACM Press, 216–223.

Fateman, R. J. 1976. The MACSYMA "big-floating-point" arithmetic system. In *Proceedings of the third ACM symposium on Symbolic and Algebraic Computation*. ACM Press, 209–213.

Fritsch, F. N., Shafer, R. E., and Crowley, W. P. 1973. Solution of the transcendental equation $we^w = x$. *Commun. ACM 16,* 2, 123–124.

Graham, R. L., Knuth, D. E., and Patashnik, O. 1994. *Concrete Mathematics.* Addison-Wesley.

Granlund, T. 2004. *GNU MP:The Gnu multiple precision arithmetic library*, 4.1.3 ed. GNU, http://www.swox.com/gmp/.

Jeffrey, D. J. 1994. The importance of being continuous. *Mathematics Magazine 67,* 4, 294–300.

Lozier, D. W. and Smith, J. M. 1981. Algorithm 567: Extended-range arithmetic and normalized legendre polynomials [a1], [c1]. *ACM Trans. Math. Softw. 7,* 1, 141–146.

Monagan, M. B., Geddes, K. O., Heal, K. M., Labahn, G., Vorkoetter, S. M., McCarron, J., and DeMarco, P. 2001. *Maple 7 Programming Guide*. Waterloo Maple, Inc.

Richardson, D. 1992. The elementary constant problem. In *ACM International Symposium on Symbolic and Algebraic Computation*. ACM Press, 108–116.

Schonfelder, J. L. 2001. Variable precision arithmetic: a Fortran 95 module. *SIGPLAN Fortran Forum 20,* 3, 2–11.

Smith, J. M., Olver, F. W. J., and Lozier, D. W. 1981. Extended-range arithmetic and normalized legendre polynomials. *ACM Trans. Math. Softw. 7,* 1, 93–105.

Tupper, J. 2001. Reliable two-dimensional graphing methods for mathematical formulae with two free variables. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM Press, 77–86.

van der Hoeven, J. 1999. Fast evaluation of holonomic functions. *Theoretical Computer Science 210,* 1, 199–215.

A: Maple code for evaluation of ω

```
#
# Interface to evalf, using a global variable, in the current fashion.
#
`evalf/Wrightomega` := proc( ) local z, zr, res;
  z := evalf( args[1] );
  if not type(z, complex(float) ) then
    return 'Wrightomega'( z ) ;
  elif type(z,embedded_real) then
    # In this case do the computation disregarding the sign of the imaginary part,
    # but put it back later.
    userinfo( 2, Wrightomega, "Calling real evalf code " );
    zr := Re(z);
    res := `evalf/Wrightomega/real`(  zr );
    # Because Im( Wrightomega( z ) ) > 0 if Im( z ) > 0, we keep the sign of zero
    # the same as the sign of zero on input. Im(z) = 0., -0., or 0 here.
    if not type(z,float) then
      res := res + Im(z)*I;
    end if;
    res;
  else
    userinfo( 2, Wrightomega, "calling complex code " );
    `evalf/Wrightomega/complex`( z );
  end if;
end proc:
#
# Details of floating-point evaluation: Real Case.
# We use variable order iteration starting with a low-accuracy initial guess,
```

```
# using low precision to start with.  Increasing precision as the accuracy
# is increased saves computation time for large settings of Digits.
# The highest order available is usually used until the last step,
# when just low enough order is used to get where we are going.
#
`evalf/Wrightomega/real` := proc( x::embedded_real )
  local Asympt_Wrightomega, cond, converged, Cs, df, ddf, errests, extras, f,
        goodEnough, guardedDigits, guards, i, inputDigits, k, nextDigits, N, oldDigits,
        predictions, predicted_residual, r, rho, tol, vp1, vp1sq, w ;


  inputDigits := Digits;
  # The iterative formulae are slightly unstable for large x, so add guard digits.
  guards := 2 + max(0, ilog10(abs(x)) );
  guardedDigits := Digits+guards;

  Digits := hDigits; # Start with hardware floats (sufficient, in real arith)

  tol := Float(1,-inputDigits);

  w := realguess( x );
  # May underflow to zero for x < -49.4 billion
  if w = 0 then
     return Maple_floats(MIN_FLOAT)
  end if;

  userinfo( 2, Wrightomega, "Initial guess is", w );

  # An initial crude estimate of the residual is ok

  r          := Re( evalf[min(2*Digits, guardedDigits)]( x - w - ln(w) ) );
  if r=0 then
     r := Re(evalf[guardedDigits](x-w-ln(w)));
     if r=0 then return evalf[inputDigits](w) end if;
  end if;

  # We compute the series coefficients roughly, so we may predict
  # the number of Digits needed as we go.
  # We use 2 more coefficients as forward error estimates.
  Cs, cond := Wrightomegaseriescoeffs( r, x, w, w+1, MaximumOrder+2, 1, Digits, true );

  extras := seq( max(ilog10( evalf[hDigits]( cond[k] )) ,0 ), k=1..MaximumOrder );

  #
  # Now the main loop using Nth order iteration
  #

  # This predicts how accurate the NEXT answer will be.
  vp1   := 1 + w;
  vp1sq := vp1^2;
  rho := r/vp1sq;
  errests := seq( evalf[hDigits]((1+w)*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
  predictions := seq( -ilog10( evalf[hDigits](abs(errests[k])) ) , k=1..MaximumOrder );

  for k from MaximumOrder by -1 to 1 while predictions[k] > guardedDigits do od;
  N := max(1,min(MaximumOrder,k+1));
```

```
predicted_residual := vp1/w*errests[N];

userinfo( 5, Wrightomega, "Predicted accuracy is ", predictions[N], N );

nextDigits := min( guardedDigits, predictions[N] ) + max(extras[N],0) ;
oldDigits := Digits;
Digits    := max( hDigits, nextDigits );

# Now start the residual correctly
r := Re( evalf[Digits+oldDigits]( x - w - ln(w) ) );

# The condition number of omega is (except at the branch points or the discontinuities,
# neither of which is relevant here for the real case) x/(1+omega).  Hence we use this
# in our convergence test---we say that the iteration has converged if the estimated
# accuracy is as good as desired.  Note that this is an estimate only, and might
# be fooled (for example, for x=0---so we have to handle that case separately).

# relative forward error = r/(1+w)
# relative backward error = r/x
converged := evalb( abs( (r/(1+w)) ) <= tol or (x<>0 and abs(r/x) <= tol) );

# Nth order ITERATION
# If an Nth order method doesn't converge in MaxIterations iterations,
# Ntupling digits all the while, then at the end we are working in kernelopts(maxdigits)
# digits (more than 268 million on my system); hence failure is clear.  But really
# we should have hit the limit of guardedDigits before then anyway.
goodEnough := false;
for i to MaxIterations while not converged do

  userinfo( 5, Wrightomega, "Using", N, "th order formula" );

  w         := WrightomegaIterate( w, x, N, +1, r, w+1 );

  userinfo( 5, Wrightomega, "After calculating the residual ", i, "time(s) using ",
                            Digits, " Digits, the estimate is ", w );
  userinfo( 5, Wrightomega, "Estimated forward accuracy of this answer", errests[N], N );

  goodEnough := evalb( abs(errests[N]) <= tol or (x<>0 and abs(predicted_residual/x) <= tol));
  if goodEnough and _EnvDoFinalResidual=false then
    converged := true;
    break
  fi; # Don't do final residual

  # Decide what order to use next time:
  rho := evalf[hDigits]( predicted_residual/vp1sq );
  errests := seq( evalf[hDigits]((1+w)*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
  predictions := seq( -ilog10( evalf[hDigits]( abs(errests[k]) ) ) , k=1..MaximumOrder );

  # find min order where predicted accuracy >= needed precision
  for k from MaximumOrder by -1 to 1 while predictions[k] > guardedDigits do od;
  N := max(1,min(MaximumOrder,k+1));

  nextDigits := min( guardedDigits, predictions[N] ) + max( extras[N], 0 );

  # Always work at least in hardware precision
  oldDigits := Digits;
```

```
    Digits    := max( hDigits, nextDigits );

    # The Re in the following statement removes spurious 0.*I's.  It should not be needed.
    # We compute the residual to N+1 times the Digits used to compute w, because the leading
    # Digits of f should all be zero.  N will usually be right from last time,
    # but is sometimes too large.  The first oldDigits of r are zero.

    r         := Re( evalf[Digits+oldDigits]( x - w - ln(w)) );

    # relative forward error = r/(1+w)
    # relative backward error = r/x
    converged := evalb( abs( r/(1+w) ) <= tol or (x<>0 and abs(r/x) <= tol) );
    rho := evalf[hDigits]( r/vp1sq );
    errests := seq( evalf[hDigits]((1+w)*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
    predicted_residual := evalf[hDigits]( (1+w)/w * errests[N] );

  end do;

  userinfo( 1, Wrightomega, "Number of iterations taken is ", i);

  if not converged then
    error "Convergence failure, input x = %1, current best estimate for Wrightomega = %2,
          residual = %3, current setting of Digits = %4, maximum permitted number of Digits = %5",
          x,w,r,Digits,guardedDigits;
  end if;

  # We clean off the guard digits before returning the answer.

  evalf[inputDigits]( w )

end proc:

# C O M P L E X   C A S E
#
# Details of floating-point evaluation: Complex Case. (based on real case)
# We use Nth order iteration starting with a low-accuracy initial guess,
# using low precision to start with.  Increasing precision as the accuracy
# is increased saves computation time for large settings of Digits.
#
`evalf/Wrightomega/complex` := proc( z::complex(float) )
  local aboveBottomCut, aboveTopCut, Asympt_Wrightomega, belowBottomCut, belowTopCut, betweenCuts,
        converged, df, ddf, i, inputDigits, f, guards, guardedDigits, nearBottomCut, nearTopCut,
        onBottomCut, onTopCut, p, pade, pi, tol, w, x, y, zpPi, zmPi, zpPip1, zmPip1, near, branchguards;

  x := Re(z);
  y := Im(z);
  pi := evalf(Pi);  # If the user inputs an imaginary part that evalf's to the same number
                    # that Pi does, at the input precision, the imaginary part is
                    # aggressively declared to be exactly pi.

  Rounding := -infinity; # We close the branches from below.
  zpPi := evalf(z+Pi*I); # Do these computations at the input precision.
  zmPi := evalf(z-Pi*I); # Otherwise, doing them later can cause branch crossings.
  zpPip1 := evalf(zpPi + 1);
  zmPip1 := evalf(zmPi + 1);
  userinfo(5,Wrightomega,"z + Pi*I = ", zpPi, "z - Pi*I = ", zmPi );
```

```
Rounding := nearest;   # Subsequent computations are done round-to-nearest.
                       # On exit, this environment variable will automatically be reset.

# Comparisons are done at the current setting of Digits.
aboveTopCut    := evalb( y >  pi );
onTopCut       := evalb( y =  pi );
belowTopCut    := evalb( y <  pi and y >= 0 );
aboveBottomCut := evalb( y > -pi and y <= 0 );
betweenCuts    := evalb( belowTopCut or aboveBottomCut );
onBottomCut    := evalb( y = -pi );
belowBottomCut := evalb( y < -pi );
near           := 1.0e-2; # Magic constant
nearTopCut     := evalb( abs(y-pi) <= near  and x <= -1 + near );
nearBottomCut  := evalb( abs(y+pi) <= near  and x <= -1 + near );



inputDigits := Digits;
# Iteration formula slightly unstable if z is large.
guards := 2 + max( 0, ilog10( abs(evalf[hDigits](z)) ) );
guardedDigits := Digits + guards;

# If we are
# close to the preimages of the branch point, then we need more figures; we take
# at most twice as many as previously thought necessary (it is a second order
# branch point).

if zmPip1 = 0 or zpPip1 = 0 then
   return evalf[inputDigits](-1)
end if;

if abs(zmPip1) < 1.0e-1 or abs(zpPip1) < 1.0e-1 then
   branchguards := -ilog10(
          min(evalf[hDigits](abs(zmPip1)),evalf[hDigits](abs(zpPip1)))
                          );
   guardedDigits := guardedDigits + min( guardedDigits+1, max(0,branchguards) );
   #print( "guarded", guardedDigits, "branch", branchguards );
else
   branchguards := 0;
end if;
zmPip1 := evalf[guardedDigits](Re(z)+1) + Im(zmPi)*I;
zpPip1 := evalf[guardedDigits](Re(z)+1) + Im(zpPi)*I;

# Starting digits: hardware floats ok, unless so near the branch
# point that we can't distinguish 1 from 1+w.

Digits := min( guardedDigits, hDigits+branchguards ) ;


tol := Float(1,-inputDigits);

w := complexguess( z, zpPip1, zmPip1, x, y,
                   aboveTopCut,
                   onTopCut,
                   belowTopCut,
                   aboveBottomCut,
```

```
                    betweenCuts,
                    onBottomCut,
                    belowBottomCut,
                    nearTopCut,
                    nearBottomCut);


  userinfo( 2, Wrightomega, "Initial guess is", w );

  # If the initial guess underflows, return a small number with a signed imaginary part

  if abs(w)=0 then return Maple_floats(MIN_FLOAT) + I*CopySign(0.0,Im(z)) end if;

  # If we are on or near the discontinuities, use the transformation v = -w to
  # reduce it to the continuous problem -v + ln(v) = zeta for zeta near the axis.
  # The initial guesses will be good enough to distinguish between the cases
  # v > 1 (corresponding to being outside the strip) and v < 1 (z inside the strip).

  # The fact that this works is gratifying.  For the analysis of why it works,
  # see the paper.

  if nearTopCut then
    return `evalf/Wrightomega/complex/regularized`( zmPi,-w, -1, inputDigits, guardedDigits, true )
  elif nearBottomCut then
    return `evalf/Wrightomega/complex/regularized`( zpPi,-w, -1, inputDigits, guardedDigits, false )
  else
    return `evalf/Wrightomega/complex/regularized`(    z, w, +1, inputDigits, guardedDigits )
  end if;

end proc:    # COMPLEX CASE
#
# Solver for s*v + ln(v) = z  , s = +/- 1
#
# Properly used, this regularizes the solution of w + ln(w) = z near the discontinuities
# at z = t +/- I*Pi, for t <= -1.
#
`evalf/Wrightomega/complex/regularized` := proc( z0, w0, s, inputDigits, guardedDigits )
   local cond, converged, df, ddf, errests, f, goodEnough, i, tol, v, w, z, vp1, vp1sq, oldDigits,
        Cs, N, r, predictions, k, nextDigits, extras, predicted_residual, rho;
   # s = +1 or s = -1
   ASSERT( evalb( s = 1 or s = -1 ));
   z   := evalf[inputDigits](   z0 );
   v   := evalf[inputDigits]( s*w0 );
   vp1 := evalf[guardedDigits]( v + 1 );
   w   := evalf[inputDigits](   w0 );
   tol    := Float(1,-inputDigits);
   # Inherit Digits from calling routine---at least hDigits
   vp1sq := vp1^2;
   #
   # Now the main loop using Nth order iteration
   # We compute the series coefficients roughly, so we may predict
   # the number of Digits needed as we go.
   # SCALED RESIDUAL predictions:
   # r[k+1] approximately (1+v[k])*(1+v[k+1])/w[k+1]*(Cs[N+1]*r[k]^(N+1)+Cs[N+2]*r[k]^(N+2))
   # predicted relative forward error approximately r[k+1]/(1+v[k+1])
   #
```

```
# Now the main loop using Nth order iteration
#
# An initial crude estimate of the residual is ok

r          := evalf[min(2*Digits, guardedDigits)]( z - v - ln(w) );

Cs, cond := Wrightomegaseriescoeffs( r, z, w, vp1, MaximumOrder+2, s, Digits, true );
extras   :=  [seq( -ilog10(evalf[hDigits](cond[k])), k=1..MaximumOrder )];
userinfo( 5, Wrightomega, "Maximum needed extra digits for instability of formula = ",
                           max(op(extras),0) );

# If the initial guess is already good enough, then quit.
if abs(r) <= tol then
   r := evalf[guardedDigits](z-v-ln(w));
   if abs(r) <= tol then
      if nargs=6 and abs(Im(w))<=tol then
         userinfo( 10, Wrightomega, "Cleaning up sign", args[6] );
         if args[6] then
            # near top cut, hence sign Im(w) should be positive
            w := s*Re(w) + I*CopySign(0.0,1);
         else
            # near bottom cut, hence sign Im(w) should be negative
            w := s*Re(w) + I*CopySign(0.0,-1);
         end if;
      else
         userinfo( 10, Wrightomega, "Signed zero need not be touched" );
         w := s*w;
      end if;
      # We clean off the guard digits before returning the answer.
      return evalf[inputDigits]( w );
   end if;
end if;

# This predicts how accurate the NEXT answer will be.
rho := r/vp1sq;
errests := seq( evalf[hDigits](vp1*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
predictions := seq( -ilog10( evalf[hDigits](abs(errests[k])) ), k=1..MaximumOrder);

for k from MaximumOrder by -1 to 1 while predictions[k] > guardedDigits do od;
N := max(1,min(MaximumOrder,k+1));
predicted_residual := evalf[hDigits]( vp1/w*errests[N] );

userinfo( 2, Wrightomega, "Initial residual is ", evalf[5](r) );
userinfo( 2, Wrightomega, "Predicted accuracy next step is ", predictions[N], N+1 );

nextDigits := min( guardedDigits, predictions[N])+ max(extras[N],0) ;
oldDigits := Digits;
Digits    := max( hDigits, nextDigits );

r := evalf[Digits+oldDigits]( z - v - ln(w) );  # Do it again at right precision.

# The condition number of Wrightomega is (except at the branch points or the discontinuities,
# z/(1+omega).  Hence we use this
# in our convergence test---we say that the iteration has converged if the estimated
# accuracy is as good as desired.  Note that this is an estimate only, and might
# be fooled (for example, for x=0---so we have to handle that case separately).
```

```
# Note that the case v=-1 exactly has already been handled, and
# by construction our residual has a factor (1+v)^2 in it.
# Thus, the branch point gives no problem.

# relative forward error = r/(1+v)
# relative backward error = r/z
converged := evalb( abs( r/vp1 ) <= tol or (z<>0 and abs(r/z) <= tol) );

# Nth order ITERATION
# If an Nth order method doesn't converge in MaxIterations iterations,
# Ntupling digits all the while, then at the end we are working in kernelopts(maxdigits)
# digits (more than 268 million on my system); hence failure is clear.  But really
# we should have hit the limit of guardedDigits before then anyway.
goodEnough := false;
for i to MaxIterations while not converged do

   userinfo( 5, Wrightomega, "Using order",N+1," formula" );

   w       := WrightomegaIterate( w, z, N, s, r, v+1 );
   v       := s*w;

   userinfo( 5, Wrightomega, "After computing the residual  ", i, "time(s) using ",
                             Digits, " Digits, the estimate is ", w );


   userinfo( 2, Wrightomega, "Predicted forward accuracy on the next step",
                              predictions[N], N+1 );

   goodEnough := evalb( abs(errests[N]) <= tol or
                        (z<>0 and abs(predicted_residual/z) <= tol ));
                        # Don't do final residual if goodEnough
   if goodEnough and _EnvDoFinalResidual=false then
      converged := true;
      break
   fi;

   rho := predicted_residual/vp1sq;
   errests := seq( evalf[hDigits](vp1*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
   predictions := seq( -ilog10( evalf[hDigits](abs(errests[k]))) , k=1..MaximumOrder);

   for k from MaximumOrder by -1 to 1 while predictions[k] > guardedDigits+1 do od;
   N := max(1,min(MaximumOrder,k+1));

   nextDigits := predictions[N]+max(extras[N],0);
   nextDigits := min( guardedDigits+max(extras[N],0), nextDigits );
   oldDigits  := Digits;
   Digits     := max( hDigits, nextDigits );

   r          := evalf[Digits+oldDigits]( z - v - ln(w) );

   userinfo( 2, Wrightomega, "Actual residual accuracy on this step ", evalf[5](r) );

   # relative forward error = r/(1+v)
   # relative backward error = r/z
   converged := evalb( abs( r/vp1 ) <= tol or (z<>0 and abs(r/z) <= tol) );
```

```
      rho := r/vp1sq;
      errests := seq( evalf[hDigits](vp1*(Cs[k+1]*rho^(k+1) + Cs[k+2]*rho^(k+2))), k=1..MaximumOrder );
      predictions := seq( -ilog10( evalf[hDigits](abs(errests[k])) ), k=1..MaximumOrder);
      predicted_residual := vp1/w*errests[N];

   end do; # main loop ends

   userinfo( 1, Wrightomega, "Number of iterations taken is ", i);

   if not converged then
      error "Convergence failure, input x = %1, current best estimate for Wrightomega = %2,"
            "residual = %3, current setting of Digits = %4, maximum permitted number of Digits = %5",
            x,w,r,Digits,guardedDigits;
   end if;

   # PROBLEM WITH SIGNED ZERO returned---needs correction
   # The iteration formula may well return (due to roundoff) the wrong
   # sign of the imaginary part; which makes verification impossible.
   # So, if the imaginary part is zero to tolerance, we replace it
   # with the correct signed zero imaginary part.

   if nargs=6 and abs(Im(w))<=tol then
      userinfo( 10, Wrightomega, "Cleaning up sign", args[6] );
      if args[6] then
         # near top cut, hence sign Im(w) should be positive
         w := s*Re(w) + I*CopySign(0.0,1);
      else
         # near bottom cut, hence sign Im(w) should be negative
         w := s*Re(w) + I*CopySign(0.0,-1);
      end if;
   else
      userinfo( 10, Wrightomega, "Signed zero need not be touched" );
      w := s*w;
   end if;
   # We clean off the guard digits before returning the answer.


   evalf[inputDigits]( w )

end proc;  # regularized


# The initial guess functions, for analysis

realguess := proc( x::embedded_real )
  local Asympt_Wrightomega, w;
  w := `evalhf/Wrightomega/cx`( array(1..2,[x,0]), evalf[hDigits](Pi), evalf[hDigits](-Pi),
                   x, 0,
                   false,
                   false,
                   true,
                   true,
                   true,
                   false,
                   false,
                   false,
```

```
                     false );

  return w[1];
end proc;  # Real guess

# Complex initial guess function (for analysis)

complexguess := proc( z::complex(float), zpPip1::complex(float), zmPip1::complex(float),
                     x, y,
                     aboveTopCut,
                     onTopCut,
                     belowTopCut,
                     aboveBottomCut,
                     betweenCuts,
                     onBottomCut,
                     belowBottomCut,
                     nearTopCut,
                     nearBottomCut )
  local w, p, pade, Asympt_Wrightomega;

  w := `evalhf/Wrightomega/cx`( array(1..2,[Re(z),Im(z)]), Im(zpPip1), Im(zmPip1),
                     x, y,
                     aboveTopCut,
                     onTopCut,
                     belowTopCut,
                     aboveBottomCut,
                     betweenCuts,
                     onBottomCut,
                     belowBottomCut,
                     nearTopCut,
                     nearBottomCut );

  return w[1]+I*w[2];
end proc;

#
# Series coefficients for Wright omega, by recursion
# Extra flags for regularized branch series (sg = -1)
#         and for computation of evaluation condition number
#         series is scaled, so y = wa + (1+sg*wa)*sum( C[k]*rho^k, k=1..N)
#         and rho = (z-a)/(1+sg*wa)^2.
#
Wrightomegaseriescoeffs := proc( r, a, wa, vp1, N::nonnegint, sgin::integer,
                                 Digin::posint, computecond::boolean )
  local cond, digs, digads, guards, C, sg, k, rho;
  userinfo(20,Wrightomega,"Computing series coefficients");
  C := array(0..N);
  Digits := Digin;
  sg := sgin;
  # We compute the scaled series coefficients, so that
  # w(a+r) = wa + vp1*sum( C[k]*(r/vp1sq)^k, k=1..infinity )
  # [This scaling does good things near the branch points, and does not
  # do any harm elsewhere.]
  C[0] := evalf( wa/vp1 );
  guards := hDigits; # Fix later
  digs := max( 0, Digits - guards );
```

```
  digads := -floor( (digs)/(N+1) );
  for k to N do
    # Use retrogressive precision
    digs := digs + digads;
    C[k] := evalf[digs+guards]( (C[k-1]*vp1 - sg*add((k-j)*C[k-j]*C[j], j=1..k-1))/k );
    #print( k, C[k] );
  end do;
  userinfo(20,Wrightomega,"Series coeffs are ", C[0], C[1], "...", C[N] );
  # The truncated series approximation has a condition number for evaluation
  # that is important in deciding how many extra digits to use for numerical
  # stability of the method.  If asked, compute the condition number of the
  # formula here, while we have the coefficients available.
  cond := NULL;
  if nargs > 7 and computecond then
    cond := array(1..N);
    Digits := hDigits;
    rho := evalf( r/(vp1)^2 );
    for k to N do
      cond[k] := evalf(abs(vp1))*add( evalf( abs( C[j]*rho^j ) ), j=1..k );
    end do;
    userinfo(20,Wrightomega,"Condition numbers are ", cond[1], cond[2], "...", cond[N] );
  end if;
  return C, cond;
end proc;


#
# Iteration formulas for solving y + ln(y) = z,
# of N+1 th order.  1 <= N <= MaximumOrder.
#
# The case y=-1 exactly is DISALLOWED in this iteration
#
# solves sg*y + ln(y) = z (sg=-1 regularizes branch cuts)
#
# RMC December 2003
WrightomegaIterate := proc ( y, z, N::posint, sg, r, vp1 )
     local C, digs, digads, i, n, rh, rho, s, t1, ysg;
     C   := array(0..N);
     rho := array(1..N);
     # We put evalf[Digits] everywhere; later these will
     # be modified for economy at high precision (not all
     # of these coefficients are needed at full precision)
     # From the series
     C := Wrightomegaseriescoeffs( z+r, z, y, vp1, N, sg, Digits, false );
     n := N;
     # Most expensive step at high Digits, assumed done OUTSIDE
     # r   := evalf[Digits]( z - ysg - ln(y) );
     # Once we have the residual, and the series coefficients,
     # computation of the iterate is simple.
     #
     ysg := y*sg;
     t1  := evalf[Digits]( r/vp1 );
     rh  := evalf[Digits]( t1/vp1 );
     digads := max( ceil(Digits/n), -ilog10(evalf[hDigits](rh)));
     # Horner's method, modified for the peculiar series
     digs := digads;
```

```
    s := evalf[digs]( C[n]*rh );
    # We use "progressive precision" here, which ought to be cheaper.
    for i from n-1 by -1 to 1 do
       digs := max(Digits, digs + digads);
       s := evalf[digs]( rh*(C[i] + s ) );
    end do;
    # Return the new estimate
    return evalf[Digits]( y + vp1*s )
end proc:
```