High Precision Computation of Elementary Functions in Maple

D.J. Jeffrey

Ontario Research Centre for Computer Algebra,
Department of Applied Mathematics,
The University of Western Ontario, London, Ontario, Canada

Abstract. Some methods being used to speed up floating-point computation in Maple are described. Specifically, we discuss the evaluation of the elementary functions square root and logarithm. We also describe the correct rounding of these evaluations. The key efficiency improvement is called *progressive precision increase*.

1 High Precision Computations in Maple

One of the advantages of computer-algebra systems for scientific computation is the easy way in which traditional approximate numerical computation can be combined with symbolic computation and with high-precision approximate numerical computation. Developments in this field have been given a further stimulus by the realization that many sets of equations coming from engineering or science are not exact, but contain coefficients that are known only approximately. Even if a system of equations is known exactly, there are enormous speed-ups possible if the analysis of the system can be shifted to the approximate numerical domain. As a result of the need for more access to high-precision computation, the support offered by Maple for this area is being improved, and the basis for some of this work is described here.

A person wanting high-precision arithmetic has many systems available in addition to the computer-algebra systems. The public domain GMP package [7] is a well known example. It is therefore worth asking at the outset whether one should devote time to developing high-precision approximate arithmetic for computer-algebra systems. There is the obvious commercial reason: customers want the functionality and do not want to learn GMP. In addition, however, the spread of approximate arithmetic into settings where traditionally exact computation was the only option, means that symbolic computation needs its own multi-precision arithmetic available within the system, not through an external call to a helper system.

The next issue that arises is connected with the fact that Maple uses a decimal-based number system, whereas GMP and other packages, including computer IEEE hardware arithmetic, use the binary system. The suggestion has therefore been made that Maple and other computer-algebra systems should strip out their present implementations of approximate arithmetic and convert to GMP. Logical and appealing though this be to supporters of GMP, there is no possibility that any commercial software system will do this within the foreseeable future. Therefore the assumption of this paper is that high-precision arithmetic needs to be developed and improved for an existing system such as Maple within the framework already in place inside that system.

We review some relevant facts about floating-point arithmetic in Maple. Maple has 2 floating-point number systems: *hardware-floats*, which are *de facto* the binary IEEE double-precision numbers currently supported by computer manufacturers, and *software-floats*, also called *Maple floats*, which are the ones that support arbitrary precision arithmetic in exact and approximate modes. Most of the elementary functions can be computed as hardware-floats. The functionality of hardware floats is derived from the underlying hardware *and* the C compilers used on the particular operating system platform to compile Maple. A visible consequence of this is seen in the evaluation of the function call `evalhf(Digits)`. Maple, in response to this function call, returns the number of decimal digits that it thinks the current hardware and software platform can reliably deliver to Maple, and hence the user. It should be noted that this number varies between the operating systems currently available. Users who have access to Maple on Windows and Unix systems might find it interesting to experiment with this function.

At any point in time, the floating-point environment in Maple is controlled by the global variable `Digits`, which controls the number of decimal digits retained in the mantissa of any (software) floating-point number. Maple's floating-point model assumes that a floating point number A is evaluated to the accuracy specified by `Digits` prior to being accepted as a parameter, even if A is actually known to more digits than the current setting. The problem then is to evaluate a given floating-point expression. When computing the value of the expression, the Maple model regards each number as being padded with zeros indefinitely. This is important for rounding.

For Maple 7, floating-point division has been sped up relative to previous Maple releases. As well as such efficiency improvements, Maple's floating point now conforms to the IEEE standards. Maple also supports a signed zero, as advocated by Kahan. This paper describes some further improvements in floating-point performance.

2 Inverse Functions and Iterative Schemes

We shall describe here the evaluation of the square root function and the natural logarithm function, but the same methods can be applied to many other functions, for example, division, nth roots, and the Lambert W function. All inverting schemes are variations on, or extensions to, Newton iteration. We therefore start by giving a uniform treatment of these schemes.

2.1 General Iteration Formulae

Consider solving the equation $f(x) = 0$, given an initial estimate x_0 for the solution. We expand $f(x)$ as a Taylor series around x_0.

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \tfrac{1}{2}(x - x_0)^2 f''(x_0) + \frac{1}{6}(x - x_0)^3 f'''(x_0) + \dots \tag{1}$$

Setting $h = x - x_0$ and assuming $f(x) = 0$, we can solve for h by either the Lagrange inversion theorem, or, equivalently, series reversion. Abbreviating $f(x_0)$ to f for clarity, we write

$$h = -\frac{1}{f'}f - \frac{f''}{2(f')^3}f^2 + \frac{3(f'')^2 - f'f'''}{6(f')^5}f^3 + \dots \tag{2}$$

The series is written as shown to emphasize that it is a series in powers of $f(x_0)$. Taking one term of this series leads to the classical Newton iteration, taking two terms leads to an iteration often named after Chebyshev [6], and taking two terms and converting into a fraction gives the classical Halley iteration. The continued fraction form of (2) is

$$h = \cfrac{-f}{f' + \cfrac{-f}{2f'/f'' + \cfrac{-f}{3f'(f'')^2/(3(f'')^2 - 2f'f''')}}}\ , \tag{3}$$

and dropping higher-order terms and reverting to standard iteration notation, we get Halley's method as

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k) - f(x_k)f''(x_k)/(2f'(x_k))}\ . \tag{4}$$

This form is better known than Chebyshev's, possibly because it looks neater. However, Chebyshev's expression requires only one division and 3 multiplies, whereas Halley's form requires 2 divisions and 1 multiplication, and therefore the speed of the formulae will depend on the relative speed of the operations. As a closing comment, the series (2) can be approximated with a lot of different forms, for example, Padé approximants, leading the many variations on the same idea [6].

2.2 Specific Computational Schemes

The formulae above have been developed for a general function f. For the particular functions square root and logarithm, specific formulae can be developed in which the coefficients can be given in more detail and more explicitly. We summarize these formulae here. All the iteration formulae give x_{n+1} in terms of x_n.

Square Root. Let x_n be an approximation to $x = A^{1/2}$. Write

$$x = \left(x_n^2 + A - x_n^2\right)^{1/2} = x_n \left(1 + A/x_n^2 - 1\right)^{1/2}$$
$$= x_n \sum_{k=0}^{\infty} \binom{1/2}{k} \left(\frac{A}{x_n^2} - 1\right)^k . \tag{5}$$

This is an example of an *expansion by residuals* according to Popov and Hare [4].

Taking any finite number of terms, we obtain an iteration formula for x_{n+1} from the exact formula for x. The first two terms reproduce the Newton iteration formula, also called Heron's rule by Kahan [5]. Taking the first three terms gives Chebyshev's method [6] or Halley's method, after conversion to a fraction as described above. Higher-order methods are obviously available by taking more terms in the series.

Markstein [1, 2] was apparently the first to propose that a faster way to compute $x = \sqrt{A}$ is to compute first the reciprocal $y = 1/\sqrt{A}$, followed by $x = Ay$. Letting y_n be an approximation to y, we write

$$y = y_n(1 + (Ay_n^2 - 1))^{-1/2} = y_n \sum_{k=0} \binom{-1/2}{k} (Ay_n^2 - 1)^k \tag{6}$$
$$= y_n \left(1 + \frac{1}{2}(1 - Ay_n^2) + \frac{3}{8}(1 - Ay_n^2)^2 + \frac{5}{16}(1 - Ay_n^2)^3 \ldots\right) . \tag{7}$$

Since the potential advantage of this series is the fact that there is no division, there is no point in forcing this into Halley's form. Notice that in either base 2 or in base 10, the coefficients can be expressed exactly as multiplying factors. It is obvious that (6) is a convergent series for $|Ay_n^2 - 1| < 1$.

Logarithm Function. In Maple, the exponential function has been programmed in the Maple kernel and therefore can be computed very quickly relative to library-defined functions. For this reason, the inversion of the exponential function is an efficient way to compute the logarithm. Let x_n be an approximation to $x = \ln A$. Then

$$x = \ln A = \ln\left[e^{x_n}\left(1 - (1 - Ae^{-x_n})\right)\right] \tag{8}$$
$$= x_n + \ln\left(1 - (1 - Ae^{-x_n})\right) = x_n - \sum_{k=1} k^{-1}(1 - Ae^{-x_n})^k . \tag{9}$$

Again, one term of the series corresponds to Newton iteration, and the series converges for all approximations x_n such that $|1 - Ae^{-x_n}| < 1$.

3 Implementation of Direct Loops Using Progressive Precision

The naïve way to program these iterative schemes is to loop until either the backward error (also called the residual) is small enough, or until the forward error is small enough, usually decided by checking whether the iterative sequence has reached a constant value. Either way, an unnecessary computation is made because the last computations merely confirm that the desired accuracy has been reached. A more efficient way is to decide in advance how many loops are needed. For this we need a forward error analysis.

3.1 Forward and Backward Error

This analysis assumes Maple software floats in base 10. Suppose $y = 1/\sqrt{A}$ is required correct to D digits. This means we need a relative forward error less than 10^{-D}.

Suppose we have some y_0 correct to d digits. This means that

$$|y - y_0|/y < \phi\, 10^{1-d} \text{ and } \phi < 1 .$$

If one requires that y_0 be accurate to d digits after rounding, then the stronger $\phi < \frac{1}{2}$ is required. The backward error (residual) in this case is

$$|Ay_0^2 - 1| \approx 2\phi 10^{1-d} . \tag{10}$$

Theorem. Given y_0 correct to $d > 1$ digits (rounded), a Newton improvement is correct to $2d-1$ digits and a Chebyshev step is correct to $3d-2$ digits.
Proof. Let $y_1 = y_0 + \frac{1}{2}y_0(1-Ay_0^2)$ be the result of the Newton step. The forward error is then

$$\frac{|y-y_1|}{y} = \frac{|y-y_0-\frac{1}{2}y_0(1-Ay_0^2)|}{y} = \frac{|2y^3-3y_0y^2+y_0^3|}{2y^3} = \frac{(y-y_0)^2(2y+y_0)}{2y^3} \ .$$

So

$$\frac{|y-y_1|}{y} \approx \frac{3(y-y_0)^2}{2y^2} \le \tfrac{3}{2}\phi^2\,10^{2-2d} = \left(\tfrac{3}{2}\phi^2\right)10^{1-(2d-1)} \ .$$

Therefore in the worst case of $\phi = 1/2$, y_1 is correct to $2d-1$ rounded digits. In the case of a result y_0 correct to d truncated digits, then y_1 is accurate to $2d-2$ digits.

After a Chebyshev step, we have $y_1 = y_0 + \frac{1}{2}y_0(1-Ay_0^2) + \frac{3}{8}y_0(1-Ay_0^2)^2$. Then

$$\frac{|y-y_1|}{y} = \frac{|y-y_0|^3(8y^2+9yy_0+3y_0^2)}{8y^5} \approx \frac{20|y-y_0|^3}{8y^3} \le \tfrac{5}{2}\phi^3\,10^{1-(3d-2)} \ .$$

3.2 Initial Estimate

An important distinction between the different applications of iterative schemes concerns the initial estimate. Most of the differences between iterative schemes, of the same order, concern the behaviour when the initial estimate is far from the fixed point. Since here we want a computational tool, we must obtain a starting estimate that is close enough for the convergence theorems above to apply.

In the Maple context, this is achieved in a simple way by transferring the responsibility for the initial estimate to the hardware floating-point subsystem. Specifically, the initial estimates for $1/\sqrt{A}$ and $\ln(A)$ are obtained from `evalhf(1/sqrt(A))` and `evalhf(ln(A))`. With these starting estimates, the differences between, say, Chebyshev and Halley iteration are minor, from the point of view of convergence.

3.3 Iteration Control Algorithm

Using the results of the previous sections, we can combine the control of the iterative loops with the control of precision, and avoid unnecessary looping. Suppose we are using Newton steps to achieve a final target of D digits of accuracy. Let the accuracy *before* the last iteration be D' digits. After the last iteration, the accuracy will be $2D'-1$, and clearly this must equal D. Therefore $D' \ge (D+1)/2$. Let the number of digits in the initial estimate be `IniDigits:=evalhf(Digits)`, then the algorithm is as follows.

1. Create an array `d[i]` of intermediate precisions, using the iteration

```
i:=1
d[i]:=D
while d[i]> IniDigits do
   i:=i+1
   d[i]:=iquo(d[i-1]+1,2)
end do
```

2. Compute the starting estimate using `y=evalhf(1/sqrt(A))`
3. Iterate to D digits using the loop[1]

```
Digits:=d[i]
while i>=1 do
   y:=y + (1/2) y(1-Ay^2)
   i:= i-1
end do
```

4. With `Digits:=D`, compute $x = Ay$.

Similar considerations give us the steps for the Chebyshev iteration. The array `d[i]` is set up using the scheme `d[i]:=iquo(d[i-1]+2,3)`.

[1] This loop will be further modified in the next section.

3.4 Fine-grain Precision Control

From the point of view of optimizing the computation, the basic effects are that a higher-order method requires fewer steps, but requires more computation per step. However, this section will show that, with a fine-grained control of precision, the relevant considerations are not the obvious ones. In fact we shall see that the total number of iterations used is not a critical factor in the overall time, but only the characteristics of the final loop.

In Maple, a reasonably accurate model of the cost of arithmetic is that two numbers with m and n digits can be multiplied in $O(mn)$ time. We combine this model with some observations concerning the required precision at each step. For the sake of definiteness, we conduct the discussion in terms of square root.

Consider a Newton step in the inverse square root iteration. We write it as a pair of equations:

$$r = 1 - Ay_n^2 \ , \tag{11}$$

$$y_{n+1} = y_n + \tfrac{1}{2} y_n r \ . \tag{12}$$

Without loss of generality, we can assume the problem has been scaled so that $0 < y < 1$. Let y_n be accurate to d digits and let y_{n+1} be accurate, after computation, to $2d - 1$ digits. As noted in (10), the residual r is a quantity $O(10^{-d})$. Clearly, the sum in (12) consists of a quantity $O(1)$ added to a quantity $O(10^{-d})$. Therefore the computation of one Newton step takes place using the following settings for `Digits`, the Maple variable describing precision.

Term	`Digits`
$r = 1 - Ay_n^2$	$2d$
$t = \frac{1}{2} y_n r$	d
$y_n + t$	$2d$

Therefore the cost of this step is dominated by the computation of r. Note that the precision used is not the final target precision D requested by the user, but the lesser precision `d[i]` that was set up at the start of the iteration. Only in the last iteration, does the working precision reach D. Therefore, the last loop, when the residual must be computed at the full precision D, determines (asymptotically) the time.

3.5 Newton Compared with Chebyshev

We consider here the final loop in the Newton and Chebyshev cases. In each case the final precision is D digits, but the final loop of a Newton step will start with `Digits:=D/2`, while the final loop of a Chebyshev step will start with `Digits:=D/3`. The costs of the multiplications are set out in the following table. Recall that the stepping formulae are:

$$\begin{aligned} & r = 1 - Ay_n^2 \ , \\ \text{Newton:}\quad & y_{n+1} = y_n + (0.5) y_n r \ , \\ \text{Chebyshev:}\quad & y_{n+1} = y_n + (0.5) y_n r + (0.375) y_n r^2 \end{aligned}$$

The multiplication costs are therefore

Newton	Term	Cost
	$t_1 = Ay_n$	$(D)(D/2)$
	$t_2 = t_1 y_n$	$(D)(D/2)$
	$r = 1 - t_2$	
	$\frac{1}{2} y_n r$	$(D/2)(D/2)$
Chebyshev	**Term**	**Cost**
	$t_1 = Ay_n$	$(D)(D/3)$
	$t_2 = t_1 y_n$	$(D)(D/3)$
	$r = 1 - t_2$	
	$t_3 = r^2$	$(D/3)(D/3)$
	$\frac{1}{2} y_n (r + t_3)$	$(D/3)(2D/3)$

Adding these cost up, we see that the Newton step costs $5D^2/4$ in multiplications, while the Chebyshev step costs D^2 in multiplications. Therefore Maple uses the 3rd-order Chebyshev method for its computation.

4 Implementation Notes

4.1 Expected Input Data

The number of digits in the input data can be quite different from the number of digits in the output value. Thus there is a need in computer algebra systems for the square root of a small, exactly known number, such as an integer. Therefore the performance of an algorithm depends upon two parameters: the size (length) of the input and the size of the the requested output. The case in which both lengths are equal is the obvious case, but the case in which the input is short is significant.

4.2 Order of Evaluation

The residual $1 - Ax^2$ will evaluate in different times, depending upon the order of evaluation. If a third-order method is being used, then at the start of an iteration, if x be d digits long, then A will be $3d$ digits long. Therefore the evaluation of x^2 gives a result of length $2d$. So the two products x^2 and $(x^2)A$ are a product of d-by-d digits followed by the product $3d$-by-$2d$. In contrast, the evaluation Ax and $(Ax)x$ is a product of $3d$-by-d followed by $3d$-by-d. So if we stay with the simple model of multiplication, then we have a cost of $7d^2$ compared with $6d^2$. This is an example of how the fine control of precision during programming can affect the speed of the code.

4.3 Timings

This section gives some brief timings of square root calculations in Maple. The timings compare Maple 6 with programs based on the current theory. Maple 6 computes square roots by starting with an approximation based on a rational approximation and then loops in the way described at the start of section 3.

The table gives the time in seconds required to compute square roots of numbers at various settings of `Digits`. In each case, the times of 1000 computations were averaged. For `Digits` < 200, the fact that the built-in routine is compiled in the kernel, but the present method is interpreted code, affects the timings significantly. Therefore, the table starts at 200 digits.

Digits	Maple 6	Present method
200	0.0012	0.0014
500	0.0045	0.0025
1000	0.014	0.0041
5000	0.32	0.024
10000	1.37	0.063

5 Rounding Problems

The IEEE standard states that the square root function should be exactly rounded. This means that we imagine that the number A is padded indefinitely with zeros, and the square root computed exactly. After this computation, we round to the required D digits.

We first consider when rounding will be difficult. The difficult cases will depend upon rounding mode. In Maple, the control variable `Rounding` can be set to `nearest, 0, infinity`.

Suppose we are calculating to D digits. For the purposes of discussion only, let A be scaled so that $\sqrt{A} \approx n$, where n is a D-digit integer. The difficult cases are those in which we have the following.

$$\texttt{Rounding:=nearest;} \quad \sqrt{A} = n + 1/2 + \epsilon$$
$$\texttt{Rounding:=infinity,0;} \quad \sqrt{A} = n + \epsilon$$

We can compute n with several guard digits, meaning we can assume $|\epsilon| < 10^{-3}$, but guard digits alone are not a complete solution. Consider the Maple session (this requires an older version of Maple)

```
> restart: Digits:=25 : Rounding:=0 ;
> evalf[23](sqrt( 1234567890123^2+1))
                    1234567890122.9999999999
```

With Rounding:=0, clearly the correct answer is 1234567890123, but the computation with guard digits implies the rounded answer 1234567890122.

The problem is to decide the sign of ϵ in the equations

$$\sqrt{A} = n + 1/2 + \epsilon \Rightarrow A = n^2 + n + 1/4 + (2n+1)\epsilon + \epsilon^2 \tag{13}$$

$$\sqrt{A} = n + \epsilon \Rightarrow A = n^2 + 2n\epsilon + \epsilon^2 \tag{14}$$

Notice that if n contains D digits, then n^2 contains $2D$ digits. However, A is known only to D digits also in the standard floating-point model. Two examples of difficult cases for the case of rounding to nearest are

$$\sqrt{0.9999999999} = 0.99999999994999999999 87500$$

$$\sqrt{90000000000000900000000000000} = 300000000000001.49999999999999999625$$

Notice in the second example that A is "known" only to 15 digits: the trailing zeros are merely to force an integer answer in line with the discussion above. For the difficult cases, the most direct approach is to compute $A - n^2 - n$, in the case of rounding to nearest, using $2D$ digits of precision. The sign of the result is the sign of epsilon. However, most of the time, a few guard digits are all that is neded for rounding. Therefore the procedure at present is to compute to $D + 4$ digits. If the guard digits fall between 9999 and 0001 and rounding is to infinity or 0, then the sign of the residual is computed using $2D$ digits. If the guard digits fall between 4999 and 5001 and the rounding is to nearest, then the sign of the residual (including the .5) is computed. Thus in 3/10000 cases the extra computation is forced.

6 Closing Remarks

Most of the discussion has been directed towards computing square roots. In the case of logarithms, the same principles apply. Some details that are important are that IEEE standards do not require exact rounding in the case of logarithm. The Maple standard promises 0.6 ULP in all elementary floating point computations and this accuracy can be obtained using several guard digits, the exact number being determined by the taste of the programmer and by internal considerations. For example, the internal storage of numbers favours 4 guard digits. The greater cost of evaluating the residual in the case of the logarithm function gives a relatively greater advantage to higher-order methods. Therefore the logarithm iteration actually uses 3 terms of the log series (9).

References

1. Markstein, P. "Computation of elementary functions on the IBM RISC System/6000 Processor", IBM Journal 1990.
2. Karp, A.H. and Markstein, P. "High precision division and square root", HP labs report 93-93-42, June 1993.
3. Karp, A.H. and Markstein, P. "High precision division and square root", ACM TOMS, **23**, 561–589, (1997).
4. Popov, B.A. and Hare, D.E.G. "Using computer algebra to construct function evaluation methods", MapleTech, **3**, 18–23, 1996.
5. Kahan, W. "Square root without division", Author's web site.
6. Varona, Juan L. "Graphic and numerical comparison between iterative methods", Math. Intelligencer, **24**(1), 37 – 46 , 2002.
7. Granlund, T. "GMP manual", `http://swox.com/gmp/`, 2002.