Accurate Hyperbolic Tangent Computation

Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA

Tel: +1 801 581 5254
FAX: +1 801 581 4148

Internet: beebe@math.utah.edu

20 April 1993
Version 1.07

Abstract

These notes for Mathematics 119 describe the Cody-Waite algorithm for accurate computation of the hyperbolic tangent, one of the simpler elementary functions.

Contents

1 Introduction **1**

2 The plan of attack **1**

3 Properties of the hyperbolic tangent **1**

4 Identifying computational regions **3**
4.1 Finding x_{small} 3
4.2 Finding x_{medium} 4
4.3 Finding x_{large} 5
4.4 The rational polynomial 6

5 Putting it all together **7**

6 Accuracy of the tanh() **computation** **8**

7 Testing the implementation of tanh() **9**

List of Figures

1 Hyperbolic tangent, $\tanh(x)$. 2
2 Computational regions for evaluating $\tanh(x)$. 4
3 Hyperbolic cosecant, $\operatorname{csch}(x)$. 10

List of Tables

1 Expected errors in polynomial evaluation of $\tanh(x)$. 8
2 Relative errors in $\tanh(x)$ on Convex C220 (ConvexOS 9.0). . . . 15
3 Relative errors in $\tanh(x)$ on DEC MicroVAX 3100 (VMS 5.4). . 15
4 Relative errors in $\tanh(x)$ on Hewlett-Packard 9000/720 (HP-UX A.B8.05). 15
5 Relative errors in $\tanh(x)$ on IBM 3090 (AIX 370). 15
6 Relative errors in $\tanh(x)$ on IBM RS/6000 (AIX 3.1). 16
7 Relative errors in $\tanh(x)$ on MIPS RC3230 (RISC/os 4.52). . . 16
8 Relative errors in $\tanh(x)$ on NeXT (Motorola 68040, Mach 2.1). 16
9 Relative errors in $\tanh(x)$ on Stardent 1520 (OS 2.2). 16
10 Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun 3 (SunOS 4.1.1, Motorola 68881 floating-point). 17
11 Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun 386i (SunOS 4.0.2, Intel 387 floating-point). 17
12 Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun SPARCstation (SunOS 4.1.1, SPARC floating-point). . . 18

1 Introduction

The classic books for the computation of elementary functions, such as those defined in the Fortran and C languages, are those by Cody and Waite [2] for basic algorithms, and Hart *et al* [3] for polynomial approximations.

In this report, we shall describe how to compute the hyperbolic tangent, normally available in Fortran as the functions **tanh**() and **dtanh**(), and in C as the double precision function **tanh**().

In a previous report [1], I have shown how the square root function can be computed accurately and rapidly, and then tested using the ELEFUNT test package developed by Cody and Waite. Familiarity with the material in that report will be helpful in following the developments here.

2 The plan of attack

Computation of the elementary functions is usually reduced to the following steps:

1. Use mathematical properties of the function to reduce the range of arguments that must be considered.
2. On the interval for which computation of the function is necessary, identify regions where different algorithms may be appropriate.
3. Given an argument, x, reduce it to the range for which computations are done, identify the region in which it lies, and apply the appropriate algorithm to compute it.
4. Compute the function value for the full range from the reduced-range value just computed.

In the third step, the function value may be approximated by a low-accuracy polynomial, and it will be necessary to apply iterative refinement to bring it up to full accuracy; that was the case for the square root computation described in [1].

3 Properties of the hyperbolic tangent

The hyperbolic functions have properties that resemble those of the trigonometric functions, and consequently, they have similar names. Here are definitions of the three basic hyperbolic functions:

$$\begin{aligned} \sinh(x) &= (\exp(x) - \exp(-x))/2 \\ \cosh(x) &= (\exp(x) + \exp(-x))/2 \end{aligned}$$

$$\begin{aligned} \tanh(x) &= \sinh(x)/\cosh(x) \\ &= (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)) \end{aligned} \tag{1}$$

Three auxiliary functions are sometimes used:

$$\begin{aligned} \operatorname{csch}(x) &= 1/\sinh(x) \\ \operatorname{sech}(x) &= 1/\cosh(x) \\ \coth(x) &= 1/\tanh(x) \end{aligned}$$

The following additional relations for the hyperbolic tangent, and for the doubled argument, will be useful to us:

$$\tanh(x) = -\tanh(-x) \tag{2}$$

$$\tanh(x) = x - x^3/3 + (2/15)x^5 + \cdots \qquad (|x| < \pi/2) \tag{3}$$

$$d\tanh(x)/dx = \operatorname{sech}(x)^2 \tag{4}$$

$$\sinh(2x) = 2\sinh(x)\cosh(x) \tag{5}$$

While $\sinh(x)$ and $\cosh(x)$ are unbounded as $x \to \infty$, $\tanh(x)$ behaves very nicely. For x large in absolute value, $|\tanh(x)| \to 1$, and for x small in absolute value, $\tanh(x) \to x$. The graph of $\tanh(x)$ is sketched in Figure 1.

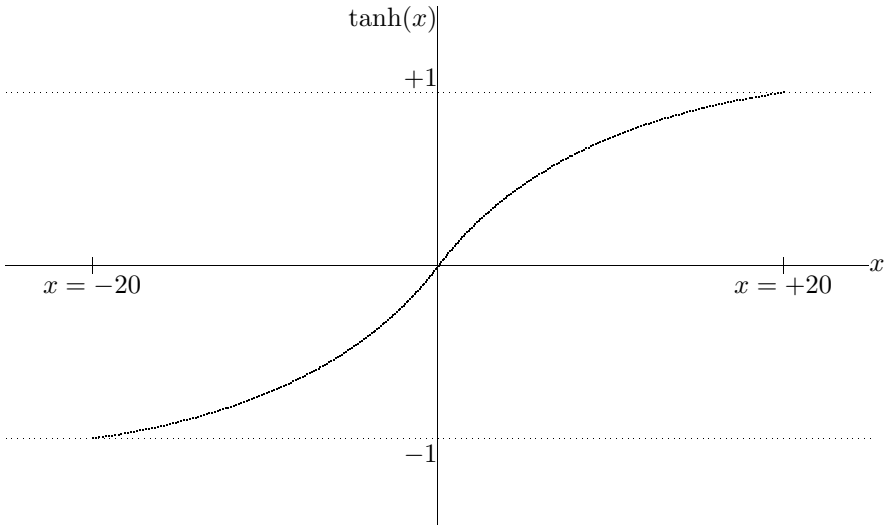

Figure 1: Hyperbolic tangent, $\tanh(x)$.

The tanh() function is useful as a data compression function, since it squashes the interval $-\infty \le x \le \infty$ onto $-1 \le x \le 1$, with relatively little distortion

for small x. For example, in preparing a mesh plot of a function with peaks of wildly-different heights, it may be useful to first take the hyperbolic tangent of the function; big peaks will still be big, but the details of the smaller peaks will be preserved.

4 Identifying computational regions

Because of equation (2) in the last section, we need only concern ourselves with positive x; for negative x, we can compute the function for $|x|$, and then just flip the sign of the computed result.

Because $\tanh(x)$ is nicely bounded for all real x, we need take no special precautions for the special case of $x = \infty$. Indeed, for relatively modest x, say $x \geq x_{\text{large}}$, we can just return $\tanh(x) = 1$; we will shortly show just where we can do this.

For somewhat smaller x values in the range $x_{\text{medium}} \leq x < x_{\text{large}}$, we can reduce equation (1) by first multiplying through by $\exp(x)$, then adding and subtracting 2 in the numerator to obtain

$$\tanh(x) \quad = \quad 1 - \frac{2}{1 + \exp(2x)}$$

In the implementation, we will actually compute this as

$$\text{temp} \quad = \quad 0.5 - \frac{1}{1 + \exp(2x)} \tag{6}$$

$$\tanh(x) \quad = \quad \text{temp} + \text{temp} \tag{7}$$

which avoids loss of leading bits on machines that have wobbling precision from bases larger than $B = 2$ (e.g. IBM mainframe).

For even smaller x in $x_{\text{small}} \leq x < x_{\text{medium}}$, the subtraction in the last equation causes accuracy loss, so we switch to an accurate rational polynomial approximation developed by Cody and Waite.

Finally, for x in $0 \leq x < x_{\text{small}}$, we will use the Taylor's series expansion, equation (3), truncated to the first term, $\tanh(x) = x$.

In summary, our computational regions are illustrated in the sketch in Figure 2.

In the next three sections, we show how to determine suitable values for the boundary values: x_{small}, x_{medium}, and x_{large}.

4.1 Finding x_{small}

Rewrite the Taylor's series, equation (3), as

$$\tanh(x) \quad = \quad x(1 - x^2/3 + (2/15)x^4 + \cdots) \qquad (|x| < \pi/2)$$

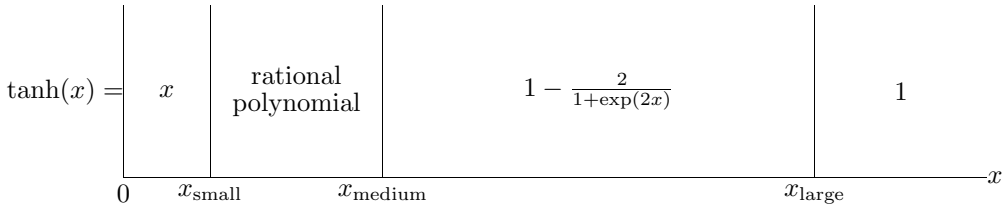

Figure 2: Computational regions for evaluating $\tanh(x)$.

and observe that as long as $1 - x^2/3 = 1$ to machine precision, then we can truncate the series to its leading term. Now, for floating-point base B with t fractional digits, the upper bound on $x^2/3$ is B^{-t-1}. To see this, consider the simple case $B = 2$ and $t = 2$. The representable numbers are then 0.00_2, 0.01_2, 0.10_2, and 0.11_2; the value 0.001_2, or B^{-3}, can be added to them without effect, since we are keeping only $t = 2$ digits. Thus, we conclude that

$$\begin{aligned} x_{\text{small}}^2/3 &= B^{-t-1} \\ x_{\text{small}} &= \sqrt{3}B^{(-t-1)/2} \end{aligned}$$

For IEEE arithmetic, we have $B = 2$, and $t = 23$ (single precision) or $t = 53$ (double precision). This gives

$$\begin{aligned} x_{\text{small}} &= \sqrt{3} \times 2^{-12} \\ &= \text{4.22863 96669 16204 32990E-04} \qquad \text{(single precision)} \\ x_{\text{small}} &= \sqrt{3} \times 2^{-27} \\ &= \text{1.29047 84139 75892 43466D-08} \qquad \text{(double precision)} \end{aligned}$$

4.2 Finding x_{medium}

The value x_{medium} is the point at which the formula in equation (6) begins to suffer loss of bits through the subtraction. This will happen for all x such that

$$\frac{2}{1 + \exp(2x)} \quad \geq \quad 1/B$$

To see why, consider again the simple case $B = 2$ and $t = 2$. The value 1 can be represented as 1.0_2, or with infinite precision, as $0.11111\ldots_2$. Loss of leading bits happens if we subtract any value of the form $0.1x_2$. There is no loss if we subtract values of the form $0.0x_2$. That is, loss happens when we subtract any value at least as large as 0.10_2, which is just $1/B$. Solving for x, we find

$$2B \quad \geq \quad 1 + \exp(2x)$$

$$\begin{aligned}(2B-1) &\geq \exp(2x)\\ \ln(2B-1) &\geq 2x\\ \ln(2B-1)/2 &\geq x\end{aligned}$$

from which we have the result we seek:

$$x_{\text{medium}} = \ln(2B-1)/2$$

Observe that this result is independent of the precision, t. For IEEE arithmetic with base $B = 2$, we find

$$\begin{aligned}x_{\text{medium}} &= \ln(3)/2\\ &= 0.54930\ 61443\ 34054\ 84570\ \text{D+00}\end{aligned}$$

4.3 Finding x_{large}

The last region boundary, x_{large}, is determined by asking: At what point will the second term in equation (6) be negligible relative to the first term? As in the determination of x_{small}, this will happen when

$$\frac{2}{1+\exp(2x)} < B^{-t-1}$$

In this region, x is large, so $\exp(2x) \gg 1$. We therefore drop the 1, and write

$$\begin{aligned}\frac{2}{\exp(2x)} &< B^{-t-1}\\ 2B^{t+1} &< \exp(2x)\\ \ln(2B^{t+1}) &< 2x\\ (\ln(2) + (t+1)\ln(B))/2 &< x\end{aligned}$$

For base $B = 2$, this simplifies to

$$(t+2)\ln(2)/2 < x$$

from which we find for IEEE arithmetic

$$\begin{aligned}x_{\text{large}} &= 12.5\ln(2)\\ &= 8.66433\ 97569\ 99316\ 36772\ \text{E+00} \quad \text{(single precision)}\\ x_{\text{large}} &= 27.5\ln(2)\\ &= 19.06154\ 74653\ 98496\ 00897\ \text{D+00} \quad \text{(double precision)}\end{aligned}$$

4.4 The rational polynomial

In the region $x_{\text{small}} \le x < x_{\text{medium}}$, Cody and Waite developed several rational polynomial approximations, $\tanh(x) \approx x + xR(x)$, of varying accuracy. For $t \le 24$, which is adequate for IEEE single precision, they use

$$\begin{aligned} g &= x^2 \\ R(x) &= gP(g)/Q(g) \\ P(g) &= p_0 + p_1 g \\ Q(g) &= q_0 + q_1 g \end{aligned}$$

where the coefficients are as follows:

$$\begin{aligned} p_0 &= -0.82377\ 28127\ \text{E+00} \\ p_1 &= -0.38310\ 10665\ \text{E-02} \\ q_0 &= 0.24713\ 19654\ \text{E+01} \\ q_1 &= 1.00000\ 00000\ \text{E+00} \end{aligned}$$

For $49 \le t \le 60$, corresponding to the double-precision IEEE case, they use

$$\begin{aligned} g &= x^2 \\ R(x) &= gP(g)/Q(g) \\ P(g) &= p_0 + p_1 g + p_2 g^2 \\ Q(g) &= q_0 + q_1 g + q_2 g^2 + q_3 g^3 \end{aligned}$$

where the coefficients are as follows:

$$\begin{aligned} p_0 &= -0.16134\ 11902\ 39962\ 28053\ \text{D+04} \\ p_1 &= -0.99225\ 92967\ 22360\ 83313\ \text{D+02} \\ p_2 &= -0.96437\ 49277\ 72254\ 69787\ \text{D+00} \\ q_0 &= 0.48402\ 35707\ 19886\ 88686\ \text{D+04} \\ q_1 &= 0.22337\ 72071\ 89623\ 12926\ \text{D+04} \\ q_2 &= 0.11274\ 47438\ 05349\ 49335\ \text{D+03} \\ q_3 &= 1.00000\ 00000\ 00000\ 00000\ \text{D+00} \end{aligned}$$

By using Horner's rule for polynomial evaluation, and observing that the high-order coefficient in $Q(g)$ is exactly one, we can simplify the generation of $R(x)$ as shown in the following code extracts:

```
*     Single precision
      g = x * x
      R = g * (p(1) * g + p(0)) / (g + q(0))
      tanh = x + x * R
```

```
*     Double precision
      g = x * x
      R = g * ((p(2) * g + p(1)) * g + p(0)) /
     &    (((g + q(2))*g + q(1)) * g + q(0))
      dtanh = x + x * R
```

It is important not to try to economize by rewriting the last expression as $x(1 + R)$; that introduces accuracy loss on machines with wobbling precision from floating-point bases larger than $B = 2$.

Cody and Waite's rational form requires four multiplications, three additions, and one division in single precision, and seven multiplications, six additions, and one division in double precision.

You might wonder, *why do we not just use the Taylor's series in this region?* Here is the explanation.

The Taylor's series terms drop by powers of two, and convergence will be slowest for the largest x value in the region, namely $x = x_{\text{medium}} \approx 0.55$. That number is about $1/2$, so each successive term in the Taylor's series will be about $1/4$ the size of the preceding term. That is, each term will have about 2 more leading zero bits than the previous term.

With $t = 53$ (IEEE double precision), we will have to sum about $53/2 = 27$ terms of the series before it has converged satisfactorily. Even if we use Horner's nested multiplication form, this will take about 30 multiplications and 30 additions, which is about five times more expensive than the rational approximation.

Also, recall that each arithmetic operation potentially introduces an error of about 0.5 base-B digit in the computed result, and that coupled with the fact that the Taylor's series terms alternate in sign, while the Cody-Waite polynomials have uniform sign, means that the Taylor's series will in practice be less accurate.

5 Putting it all together

We now have all of the pieces that we need to develop a program for the evaluation of the hyperbolic tangent.

Our program will first check for the case of a negative argument, so that we can use the inversion formula, equation (2), to allow computation only for non-negative x. Then we need to check for the possibility that x is an IEEE NaN, in which case, we must generate a run-time NaN that the caller can trap. Following that, we simply fill in the remaining branches of a block-IF statement with the code for each of the four regions we identified.

For the time being, this job is being left as an assignment for the class to complete.

6 Accuracy of the tanh() computation

In the regions $0 \le x < x_{\text{small}}$ and $x_{\text{large}} \le x \le \infty$, tanh() reduces to either x or 1, and so may be regarded as exact.

In the rational polynomial region, $x_{\text{small}} \le x < x_{\text{medium}}$, accuracy is limited by the accuracy of the rational polynomial. Cody and Waite's approximation is good to $t = 60$ bits, which is 7 more than we have in IEEE double precision, so in that region too the error is expected to be quite small, arising only from the few arithmetic operations needed to construct the rational approximation.

With the default rounding of IEEE 754 arithmetic, the average error of a binary floating-point operation is zero bits, and the maximum error is 0.5 bits. With truncating arithmetic, such as that of IBM mainframes, the average error is 0.5 bits, and the maximum error, 1 bit. From the code fragments on page 6, the single precision computation takes eight operations, and the double-precision computation, fourteen operations. The expected errors are summarized in Table 1. The differences between rounding and truncating arithmetic are striking evidence of why rounding is generally preferred.

Table 1: Expected errors in polynomial evaluation of $\tanh(x)$.

precision	arithmetic	average error (bits)	maximum error (bits)
single	rounding	0	4
	truncating	4	8
double	rounding	0	7
	truncating	7	14

The third region, $x_{\text{medium}} \le x < x_{\text{large}}$, is the questionable one. Here, we use the native Fortran **exp**() or **dexp**() functions, or the C **exp**() function, for the evaluation of $\exp(2x)$; what happens if those functions are inaccurate?

The discussion in [1] showed that the relative error in the computed function, dy/y, is related to the error in the function argument, dx/x, by

$$\frac{dy}{y} = x\frac{f'(x)}{f(x)}\frac{dx}{x}$$

where $xf'(x)/f(x)$ is a *magnification* factor. This relation is derived purely from calculus; it is *independent* of the precision with which we compute numerical approximations to the function.

For the case of the square root function, the magnification factor is a constant, 1/2, so argument errors are diminished in the function value.

On the other hand, for the $\exp(x)$ function, the magnification factor is x, which says that for large x, the computed exponential function will have a large

relative error, and the *only way* that error can be reduced is to use higher precision. That may be difficult if we are already using the highest-available floating-point precision; use of a multiple-precision software floating-point emulator would be necessary, and slow. With existing packages, software emulation of floating-point arithmetic is usually about 100 times slower than hardware arithmetic.

Now, what about our $\tanh(x)$ function? We want to find dy/y in terms of the relative error in $\exp(2x)$. Let $z = \exp(2x)$, so that

$$\begin{aligned} dz/dx &= 2\exp(2x) \\ &= 2z \\ dz/z &= 2dx \end{aligned}$$

Then from equation (4) and equation (5),

$$\begin{aligned} d\tanh(x)/dx &= 1/\cosh(x)^2 \\ d\tanh(x)/\tanh(x) &= (1/(\cosh(x)^2\tanh(x)))dx \\ &= (\cosh(x)/(\cosh(x)^2\sinh(x)))dx \\ &= (1/(\sinh(x)\cosh(x)))dx \\ &= (2/(2\sinh(x)\cosh(x)))dx \\ &= (2/\sinh(2x))dx \\ &= (1/\sinh(2x))dz/z \\ &= \operatorname{csch}(2x)dz/z \end{aligned}$$

The last equation is the result we are seeking: the magnification of the relative error in $\tanh(x)$ from the relative error in the exponential function, dz/z, is $\operatorname{csch}(2x)$.

The $\operatorname{csch}(x)$ function looks something like the sketch in Figure 3. Since $\operatorname{csch}(x)$ rises in absolute value as the origin is approached, the largest magnification factor will happen for the left end of the interval, $x = x_{\text{medium}} = 0.55$. At that point, $\operatorname{csch}(2 \times 0.55) = 0.75$; at $x = 2$, $\operatorname{csch}(4) = 0.036$; at $x = 5$, $\operatorname{csch}(10) = 9.0\text{D-}05$, and at the right-end of the interval, $\operatorname{csch}(38.1) = 5.7\text{D-}17$.

In other words, the magnification factor is *never* larger than 1, and falls very rapidly toward zero. Thus, over most of this range, modest errors in the exponential function will have little effect on the computed result.

In summary, we expect that our implementation of tanh() should be accurate over the entire real axis with an average error of only 1 or 2 bits in the last place.

7 Testing the implementation of tanh()

The Cody-Waite ELEFUNT package contains test programs that can be used to test the accuracy of our $\tanh(x)$ implementation in both single and double precision.

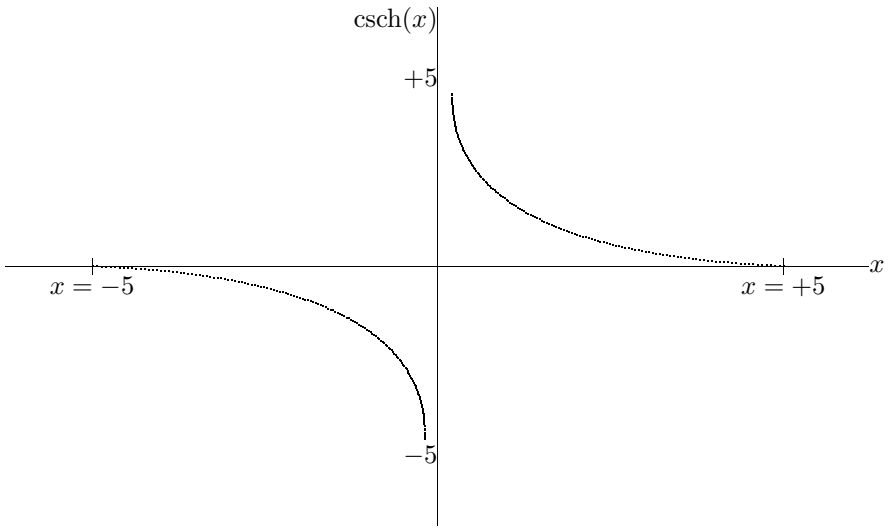

Figure 3: Hyperbolic cosecant, csch(x).

Those programs expect to test the built-in Fortran functions, **tanh**() and **dtanh**(), or for the C version, **tanh**(). If we wish to use them to test our Fortran implementation of these functions, we need to make a minor modification to inform the Fortran compiler that references to **tanh**() and **dtanh**() are to private versions of those routines. In the single-precision test program, we need to insert the statements

```
REAL TANH
EXTERNAL TANH
```

and in the double-precision test program, we need to add

```
DOUBLE PRECISION DTANH
EXTERNAL DTANH
```

This will have already been done in the test programs supplied to the class. No changes are required in the C version of the test program, because the C language does not have intrinsic functions that are treated specially by the compiler.

Here is the output of the test program for the Fortran **dtanh**() on the Sun 4 (SunOS 4.0.3):

```
TEST OF DTANH(X) VS (DTANH(X-1/8)+DTANH(1/8))/(1+DTANH(X-1/8)DTANH(1/8))
```

```
   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.1250D+00,     0.5493D+00)


 DTANH(X) WAS LARGER   633 TIMES,
              AGREED   839 TIMES, AND
         WAS SMALLER   528 TIMES.


 THERE ARE  53 BASE   2 SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER


 THE MAXIMUM RELATIVE ERROR OF     0.4867D-15 =    2 ** -50.87
    OCCURRED FOR X =     0.356570D+00
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   2.13


 THE ROOT MEAN SQUARE RELATIVE ERROR WAS     0.1430D-15 =    2 ** -52.64
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.36


TEST OF DTANH(X) VS (DTANH(X-1/8)+DTANH(1/8))/(1+DTANH(X-1/8)DTANH(1/8))


   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.6743D+00,     0.1941D+02)


 DTANH(X) WAS LARGER   575 TIMES,
              AGREED   933 TIMES, AND
         WAS SMALLER   492 TIMES.


 THERE ARE  53 BASE   2 SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER


 THE MAXIMUM RELATIVE ERROR OF     0.3249D-15 =    2 ** -51.45
    OCCURRED FOR X =     0.835594D+00
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.55


 THE ROOT MEAN SQUARE RELATIVE ERROR WAS     0.9295D-16 =    2 ** -53.26
 THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00


SPECIAL TESTS
```

```
THE IDENTITY   DTANH(-X) = -DTANH(X)   WILL BE TESTED.

       X          F(X) + F(-X)

 0.5556857D-01  0.00000000000000000D+00

 0.2588943D+00  0.00000000000000000D+00

 0.2233350D+00  0.00000000000000000D+00

 0.6093917D+00  0.00000000000000000D+00

 0.7458729D+00  0.00000000000000000D+00

THE IDENTITY DTANH(X) = X , X SMALL, WILL BE TESTED.

       X          X - F(X)

 0.2929164D-16  0.00000000000000000D+00

 0.1464582D-16  0.00000000000000000D+00

 0.7322910D-17  0.00000000000000000D+00

 0.3661455D-17  0.00000000000000000D+00

 0.1830728D-17  0.00000000000000000D+00

THE IDENTITY DTANH(X) = 1 , X LARGE, WILL BE TESTED.

       X          1 - F(X)

 0.1940812D+02  0.00000000000000000D+00

 0.2340812D+02  0.00000000000000000D+00

 0.2740812D+02  0.00000000000000000D+00

 0.3140812D+02  0.00000000000000000D+00

 0.3540812D+02  0.00000000000000000D+00

TEST OF UNDERFLOW FOR VERY SMALL ARGUMENT.

      DTANH( 0.331389-242) = 0.331389-242
```

```
 THE FUNCTION DTANH WILL BE CALLED WITH THE ARGUMENTInf
       DTANH(Inf          ) = 0.100000D+01

 THE FUNCTION DTANH WILL BE CALLED WITH THE ARGUMENT  0.4940656-323
       DTANH( 0.494066-323) = 0.494066-323

 THE FUNCTION DTANH WILL BE CALLED WITH THE ARGUMENT  0.0000000D+00
       DTANH( 0.000000D+00) = 0.000000D+00

 THIS CONCLUDES THE TESTS
```

The student implementation of this function should be able to get very similar results; the author's actually did slightly better than the Sun **dtanh**().

The Sun implementation of Fortran provides a run-time library function that can control the IEEE 754 rounding direction. By the simple addition of the statements

```
character*1 out
integer ieeer, ieee_flags
ieeer = ieee_flags ( 'set', 'direction', 'tozero', out )
if (ieeer .ne. 0) print *,'ieee_flags() error return'
```

to the beginning of the test program, the calculations can be performed with rounding to zero, that is, truncation. The `'tozero'` argument can be changed to `'positive'` for rounding upward to $+\infty$, to `'negative'` for rounding upward to $-\infty$, and to `'nearest'` for the default rounding.

A comparison of the measured maximum relative error bit losses for each of these choices is given in Tables 2–12 for several different architectures.

Although the effect of truncating arithmetic in the rational polynomial region is evident, it is by no means as big as our pessimistic worst-case estimates given earlier on page 8.

It is curious, and unexpected, that rounding-to-nearest gives larger errors in the exponential region on all of the Sun systems.

On the IBM PC, the Microsoft C, Borland Turbo C, and TopSpeed C compilers all offer a library function, `_control87()`, for programmer control of IEEE 754 floating-point rounding modes. Lahey Fortran provides no such facility. However, it can call Microsoft and Turbo C functions and assembly code routines, so the facility could be added.

On larger systems, Sun may be unique in offering the Fortran programmer control over the rounding mode in Motorola 68xxx (Sun 3), Intel 386 (Sun 386i), and SPARC (Sun 4) CPUs.

The Convex C1 and C220 systems offer only a partial implementation of IEEE arithmetic, with neither infinity, nor NaN, nor gradual underflow, nor

rounding-mode control. The Convex native-mode arithmetic is identical to that of the DEC VAX architecture.

The 32-bit DEC VAX processors have a single-precision format that resembles IEEE 754, without gradual underflow or infinity. There is a floating-point reserved operand that is somewhat like a NaN, but always causes a trap. There are two formats of double precision, D-floating, with the same exponent range as in single precision, and G-floating, with a range comparable to IEEE 754 64-bit precision. The VAX architecture also defines a quadruple-precision 128-bit floating-point arithmetic, called H-floating. In the smaller VAX models, this is implemented entirely in software.

The Hewlett-Packard PA-RISC architecture on the HP 9000/7xx and 9000/8xx machines supports all four founding IEEE 754 modes, but neither the compilers nor the run-time library make them accessible to the Fortran or C programmer.

The IBM 3090 mainframe floating-point arithmetic is truncating, and has wobbling precision. There is no support for other rounding modes.

The IBM RS/6000 and coming DEC Alpha processors provide only compile-time choice of rounding modes.

Although the MIPS CPUs used in DECstation and Silicon Graphics workstations have user-settable rounding modes, neither vendor provides either a compiler option or library support for doing so. MIPS' own workstations have a manual page describing a C run-time library routine, `fpsetround()`, for setting the floating-point mode, but neither it nor its associated include file could be found by the compiler and linker.

Like the Sun 3, the NeXT uses the Motorola floating-point processor, which uses 80-bit arithmetic. The longer precision likely accounts for the low errors in Table 8, although the same behavior is not seen in the Sun 3 results in Table 10.

The Stardent 1520 hardware implements only round-to-nearest, and gradual underflow is not supported.

References

[1] Nelson H. F. Beebe. Accurate square root computation. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah, Salt Lake City, UT 84112, USA, November 29 1990.

[2] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1980. ISBN 0-13-822064-6. x + 269 pp. LCCN QA331 .C635 1980.

[3] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thatcher, Jr., and Christoph Witzgall. *Computer Approximations*. Robert E. Krieger Publishing Company, Hunt-

Table 2: Relative errors in $\tanh(x)$ on Convex C220 (ConvexOS 9.0).

region	MRE	RMS
	bit loss	
polynomial	1.60	0.00
exponential	1.55	0.00

Table 3: Relative errors in $\tanh(x)$ on DEC MicroVAX 3100 (VMS 5.4).

arithmetic	region	MRE	RMS
		bit loss	
D-floating	polynomial	1.81	0.00
G-floating	polynomial	1.37	0.00
D-floating	exponential	2.00	0.00
G-floating	exponential	1.16	0.00

Table 4: Relative errors in $\tanh(x)$ on Hewlett-Packard 9000/720 (HP-UX A.B8.05).

region	MRE	RMS
	bit loss	
polynomial	1.99	0.06
exponential	1.67	0.00

Table 5: Relative errors in $\tanh(x)$ on IBM 3090 (AIX 370). This system offers only truncating floating-point arithmetic, equivalent to round-to-zero. Double precision has 14 base-16 digits, equivalent to 53–56 bits.

region	MRE	RMS
	bit loss	
polynomial	1.05	0.73
exponential	1.00	0.63

Table 6: Relative errors in $\tanh(x)$ on IBM RS/6000 (AIX 3.1).

region	rounding direction	MRE	RMS
		bit loss	
polynomial	`'tonearest'`	1.30	0.00
	`'tozero'`	1.99	0.05
	`'negative'`	1.99	0.05
	`'positive'`	1.62	0.00
exponential	`'tonearest'`	1.55	0.00
	`'tozero'`	1.67	0.00
	`'negative'`	1.67	0.00
	`'positive'`	1.55	0.00

Table 7: Relative errors in $\tanh(x)$ on MIPS RC3230 (RISC/os 4.52).

region	MRE	RMS
	bit loss	
polynomial	1.99	0.06
exponential	1.67	0.00

Table 8: Relative errors in $\tanh(x)$ on NeXT (Motorola 68040, Mach 2.1).

region	MRE	RMS
	bit loss	
polynomial	0.35	0.00
exponential	0.07	0.00

Table 9: Relative errors in $\tanh(x)$ on Stardent 1520 (OS 2.2). This system offers only round-to-nearest floating-point arithmetic.

region	MRE	RMS
	bit loss	
polynomial	1.99	0.06
exponential	1.58	0.00

Table 10: Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun 3 (SunOS 4.1.1, Motorola 68881 floating-point).

region	rounding direction	MRE bit loss
polynomial	`'tonearest'`	1.00
	`'tozero'`	0.99
	`'negative'`	0.99
	`'positive'`	1.00
exponential	`'tonearest'`	0.76
	`'tozero'`	0.73
	`'negative'`	0.73
	`'positive'`	0.74

Table 11: Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun 386i (SunOS 4.0.2, Intel 387 floating-point).

region	rounding direction	MRE bit loss
polynomial	`'tonearest'`	1.00
	`'tozero'`	1.00
	`'negative'`	1.00
	`'positive'`	1.00
exponential	`'tonearest'`	0.76
	`'tozero'`	0.73
	`'negative'`	0.73
	`'positive'`	0.73

Table 12: Effect of rounding direction on maximum relative error in $\tanh(x)$ on Sun SPARCstation (SunOS 4.1.1, SPARC floating-point).

region	rounding direction	MRE bit loss
polynomial	`'tonearest'`	1.31
	`'tozero'`	1.82
	`'negative'`	1.54
	`'positive'`	1.79
exponential	`'tonearest'`	1.55
	`'tozero'`	1.28
	`'negative'`	1.28
	`'positive'`	1.50

ington, NY, USA, 1968. ISBN 0-88275-642-7. x + 343 pp. LCCN QA 297 C64 1978. Reprinted 1978 with corrections.