Implementing Cordic Algorithms

A single compact routine for computing transcendental functions

Pitts Jarvis

Efficiently computing sines, cosines, and other transcendental functions is a process about which many programmers are blissfully ignorant. When these values are called for in a graphics or CAD program, we usually rely on a call to the compiler's run-time library. The library either derives the necessary values in some mysterious manner or calls the floating-point coprocessor to assist in the task.

The CORDIC (COordinate, Rotation DIgital Computer) family of algorithms is an elegant, efficient, and compact way to compute sines, cosines, exponentials, logarithms, and associated transcendental functions using one core routine. These truly remarkable algorithms compute these functions with n bits of accuracy in n iterations — where each iteration requires only a small number of shifts and additions. Furthermore, these routines use only fixed-point arithmetic. Using these algorithms, you can cast your entire graphics application into fixed-point, and thus avoid the cost of run-time conversion from fixed- to floating-point representation and back.

Even if you don't plan on recasting your application into fixed-point, you just might be curious how your floating-point coprocessor works. The Intel numerics family (8087, 80287, and 80387) all use Cordic algorithms, in a form slightly different than described here, to compute circular functions. The Intel implementations are described by R. Nave[1] and A. K. Yuen[2].

The implementations may be contemporary, but the algorithms are not new. J. E. Volder[3] coined the name in 1959. He applied these algorithms to build a special-purpose digital computer for real-time airborne navigation. D. S. Cochran[4] identifies their use in the HP-35 calculator in 1972 to calculate the transcendental functions.

Pitts Jarvis is a senior engineer at 3Com Corporation. He can be reached at 1275 Martin Ave., Palo Alto, CA 94301.

Mathematical Manipulation

If we have a vector $[x, y]$, we can rotate it through an angle a by multiplying it by the matrix Ra, defined in Example 1(a). Explicitly doing the matrix multiplication yields the equation in Example 1(b).

If we choose $x = 1$ and $y = 0$ and multiply that vector by Ra we are left with the vector $[\cos a, \sin a]$.

Multiplying by two successive rotation matrices, Ra and Rb rotates the vector through the angle $a + b$, or more formally $RaRb = Ra+b$. If we choose to represent the angle a as a sum of angles a_i for $i = 0$ through n (see Example 1(c)), then we can rotate the vector through the angle a by multiplying a series of rotation matrices Ra_0, Ra_1, . . . Ra_n.

By picking the a_i carefully, we can simplify the arithmetic. Notice that we can rewrite the rotation matrix by factoring out cos a as shown in Example 1(d). If we pick a_i such that $\tan a_i = 2^{-i}$ for $i = 0$ through n, all of the multiplications by tan a_i become right shifts by i bits.

Now we need to specify an algorithm so that we can represent a as the sum of the a_i. Initialize a variable, z, to a. This z will be a residue quantity, which we are trying to drive to zero by adding or subtracting a_i at the i-th step. At the first step, $i = 0$. At the i-th step, if $z \geq 0$ then subtract a_i from z. Otherwise add a_i to z. At the last step $i = n$, and z is the error in our representation of a. Notice that in Example 1(e), for large i, each additional step yields one more bit of accuracy in our representation of a.

Figure 1 shows the relative magnitudes of the incremental angles, a_i. Figure 2 gives an example of the convergence process with an initial angle of 0.65. Notice that successive iterations do not necessarily reduce the absolute error in the representation of the angle. Also notice that the error does not oscillate about zero.

At each step as we decompose a into the sum or difference of the a_i's we could also multiply our vector $[x, y]$ by the appropriate Ra_i or $R\text{-}a_i$ depending on whether we add or subtract a_i. Remember, these multiplications are nothing more than shifts. We must also multiply in the still embarrassing factor cos a_i. However, cosine is an even

function and has the property that $\cos a_i = \cos(-a_i)$. It does not matter whether we add or subtract the angle — we always multiply by the same factor! Because all of the $\cos a_i$ can be factored out and grouped together, we can treat their product as a constant and compute it only once, along with all the $a_i = \tan^{-1} 2^{-i}$.

Not all angles can be represented as the sum of a_i. There is a domain of convergence outside of which we cannot reduce the angle to within a_{n-1} of zero. See Example 1(f). For the algorithm to work, we must start with a such that $|a| \le a_{max} \approx 1.74$. This conveniently falls just outside the first quadrant. If we are given an angle outside the first quadrant, we can scale it by dividing by $\pi/2$ obtaining a quotient Q and a remainder D where $|D| < \pi/2 < a_{max}$. Since the algorithm computes both sine and cosine of D, we pick the appropriate value and sign depending on the value of Q.

What about angles within the domain of convergence? It's not obvious that the strange set we've picked (see Example 1(g)) can represent all angles within the domain of convergence to within a_{n-1}. But, using mathematical induction, Walther[5] proves that the scheme works.

The Circular Functions

One variation of the Cordic algorithm computes the circular functions — sin, cos, tan, and so on. This algorithm is shown in pseudocode in Example 2(a).

First, start with $[x,y,z]$. The x and y are as before. z is the quantity that we drive to zero, with an initial value of angle a. The first step in the loop decides whether to add or subtract a_i from the residue z. The variable s is correspondingly positive or negative. The second step reduces the magnitude of z and effects the multiplications by the tan a_i. The expression $y>>i$ means shift y right by i bits.

When you start the algorithm with $[x,y,z]$ and then drive z to zero as specified by Example 2(a), we are left with the quantities in Example 3(a). Where K is a constant, it is just the product of the cos a_i, as in Example 3(b).

For the curious, $K \approx 0.607$. The value of K can be precomputed by setting $[x,y,z]$ to $[1, 0, 0]$ and running the algorithm as before. The result is shown in Example 3(c). Take the reciprocal of the final x and we have K. Therefore, to compute sin a and cos a, set $[x,y,z]$ to $[K, 0, a]$ and run the algorithm. Example 3(d) shows the result. In effect, we start with a vector $[x, y]$, and rotate it through a given angle a, by driving z to zero. Running the algorithm with the special case where the vector initially lies along the x axis and is of length K, rotates the vector by angle a and leaves behind cos a and sin a. This relationship is shown in Figure 3.

To compute $\tan^{-1} a$, instead of z, we could choose to drive y to zero. Driv-

(a) $$R_a = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix}$$

(b) $$R_a \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cos a - y \sin a \\ x \sin a + y \cos a \end{bmatrix}$$

(c) $$a = a_0 + a_1 + \ldots + a_n$$

(d) $$R_a = \cos a \begin{bmatrix} 1 & -\tan a \\ \tan a & 1 \end{bmatrix}$$

(e) $$a_i = \tan^{-1} 2^{-i} \approx 2^{-i}, \text{ for large } i$$

(f) $$a_{max} = \tan^{-1} 2^{-0} + \tan^{-1} 2^{-1} + \ldots + \tan^{-1} 2^{-n}$$

(g) $$\pm \tan^{-1} 2^{-0} \pm \tan^{-1} 2^{-1} \pm \ldots \pm \tan^{-1} 2^{-n}$$

Example 1: *Mathematical expressions*

```
(a)  for i from 0 to n do
     {
          if (z ≥0) then s:= 1 else s:= -1;
          [x,y,z] := [x -s y>>i,  y + s x>>i, z - s a_i]
     }.
(b)  for i from 0 to n do
     {
          if (y ≥ 0) then s:= 1 else s:= -1;
          [x,y,z]:=[x + s y>>i, y - s x >>i,z +  s a_i]
     }
```

Example 2: *(a) The basic algorithm; (b) the inverse algorithm*

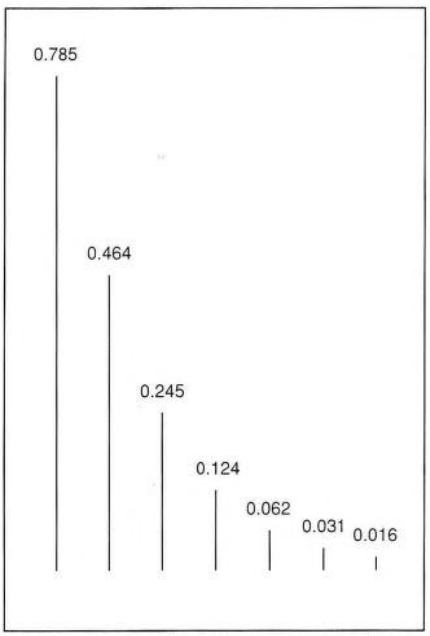

Figure 1: *$\tan^{-1} 2^{-i}$ for i=0,1,2,3,4,5,6*

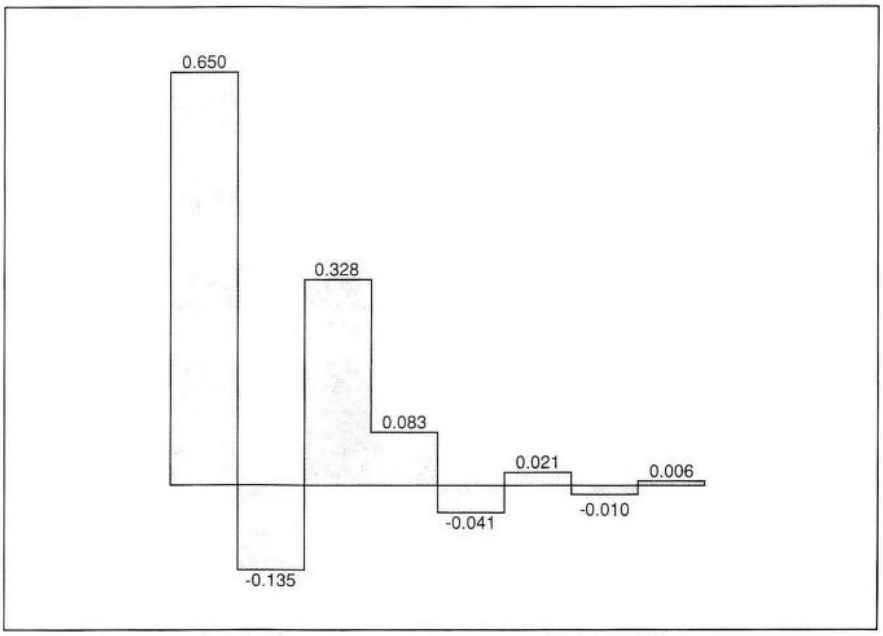

Figure 2: *Example of convergence with an initial angle of 0.65*

ing y to zero rotates the vector through the angle a, the angle subtended by the vector and the x axis, leaving the vector lying along the x axis. Start with the vector anywhere in the first or fourth quadrant and an initial value of zero in z. The first or fourth quadrant is used because almost all vectors in the second or third quadrant will not converge. At the i-th step, if $y \geq 0$, the vector lies in the first quadrant, subtract a_i from z. Move the vector closer to the x axis; rotate it by negative a_i by multiplying by the rotation matrix $R\text{-}a_i$. If $y < 0$, the vector lies in the fourth quadrant, add a_i to z and multiply the vector by Ra_i. At the end, z has the negative of the angle of the original vector $[x, y]$, $\tan^{-1} y/x = \tan^{-1} a$.

Interesting special cases include exponential, square root, and logarithm

Changing the sign of a_i has no effect on the computed values of x and y and leaves the original angle a in z rather than its negative. With this change, the inverse algorithm to drive y to zero becomes the expression shown as in the algorithm in Example 2(b).

Starting with $[x,y,z]$ and then driving y to zero using the inverse algorithm leaves behind the quantities in Example 3(e).

Hyperbolic Functions

The hyperbolic functions (sinh, cosh, and so on) are similar to the circular functions. The correspondences between these two types of functions are shown in Table 1.

By analogy, use H_a as the rotation matrix and represent a using the set $a_i = \tanh^{-1} 2^{-i}$ for $i = 1$ to n. Notice that for hyperbolics, a_0 is infinity.

Given the change in the a_i, can we still represent any angle a within the domain of convergence the same way we did for the circular functions? Unfortunately, the answer is no! Walther points out that repeating an occasional term makes the representation converge in the hyperbolic case. Repeating the terms as shown in Example 4(a) does the trick.

Except for the repeated terms and some changes of sign, the algorithms for hyperbolic functions are identical to the circular functions. Listing One,

(a) $[x,y,z] \rightarrow [\frac{1}{K}(x\cos z - y\sin z), \frac{1}{K}(y\cos z + x\sin z), 0]$

(b) $K = \cos a_0 \cdot \cos a_1 \cdot \ldots \cdot \cos a_n$

(c) $[1,0,0] \rightarrow [\frac{1}{K}, 0, 0]$

(d) $[K,0,a] \rightarrow [\cos a, \sin a, 0]$

(e) $[x,y,z] \rightarrow [\frac{1}{K}\sqrt{x^2+y^2}, 0, z + \tan^{-1}\frac{y}{x}]$

Example 3: *Circular function quantities*

(a) $a_4, a_{13}, a_{40}, a_{121}, \ldots, a_k, a_{3k+1}, \ldots$

(b) $[x,y,z] \rightarrow [\frac{1}{K}(x\cosh z + y\sinh z), \frac{1}{K}(\cosh z + x\sinh z), 0]$

(c) $[x,y,z] \rightarrow [\frac{1}{K}\sqrt{x^2-y^2}, 0, z + \tanh^{-1}\frac{y}{x}]$

(d) $[K,K,a] \rightarrow [e^a, e^a, 0]$

(e) $[a + \frac{K^2}{4}, a - \frac{K^2}{4}, \frac{1}{2}\ln\frac{K^2}{4}] \rightarrow [\sqrt{a}, 0, \frac{1}{2}\ln a]$

Example 4: *Hyperbolic function quantities*

(a) $\{\tan^{-1} 1, \tan^{-1}\frac{1}{2}, \tan^{-1}\frac{1}{4}, \ldots\}$

(b) $\{\tanh^{-1}\frac{1}{2}, \tanh^{-1}\frac{1}{4}, \ldots\}$

(c) $\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \ldots$ for $x^2 \leq 1$

(d) $\tanh^{-1} x = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} + \ldots$ for $x^2 < 1$

(e) $a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$

(f) $(\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$

Example 5: *Quantities for calculating constants*

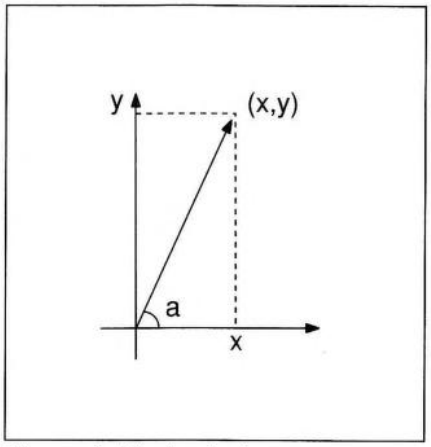

Figure 3: *The basic elements*

Hyperbolic Function	Circular Function
$\cosh x = \frac{e^x + e^{-x}}{2}$	$\cos x = \frac{e^{ix} + e^{-ix}}{2}$
$\sinh x = \frac{e^x - e^{-x}}{2}$	$\sin x = \frac{e^{ix} - e^{-ix}}{2i}$
$H_a = \begin{bmatrix}\cosh a & \sinh a \\ \sinh a & \cosh a\end{bmatrix}$	$R_a = \begin{bmatrix}\cos a & -\sin a \\ \sin a & \cos a\end{bmatrix}$
$H_a H_b = H_{a+b}$	$R_a R_b = R_{a+b}$
$e^x = \cosh x + \sinh x$	
$\ln x = 2\tanh^{-1}\frac{x-1}{x+1}$	

Table 1: *Hyperbolic functions*

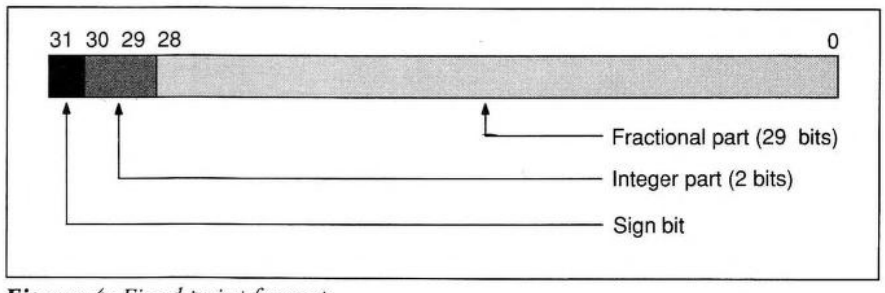

Figure 4: *Fixed point format*

(continued from page 154)
page 157, shows this in detail.

For hyperbolic functions, we start with $[x,y,z]$ and then drive z to zero. This yields the quantities in Example 4(b). Starting with $[x,y,z]$ and then driving y to zero gives the quantity shown in Example 4(c). For hyperbolics, $K \approx 1.21$.

Some interesting special cases include the exponential, square root, and natural logarithm. The exponential case is in Example 4(d) while the square root and logarithm cases are in Example 4(e).

Calculating the Constants

The algorithm to compute the circular and hyperbolic functions requires several precomputed constants. These include the scaling constant, K, for both circular and hyperbolic functions, and the sets shown in Example 5(a) and 5(b), respectively. Listing One illustrates this.

The program, written in C, uses fixed point arithmetic for all calculations. All constants and variables used to calculate functions are declared as the type *long*. The code assumes that a *long* is at least 32 bits. I have decided to represent numbers in the range $-4 \le x < 4$; this lets me represent e as a fixed point number. The high order bit is for the sign. The low order *fractionBits* (a constant defined as 29) bits hold the fractional part of the number. The remainder of the bits between the sign bit and the fractional part hold the integer part of the number. Figure 4 shows the fixed point format in graphic form.

I use power series to calculate the incremental angles a_i, as shown in Example 5(c) and 5(d), respectively. How do we know the number of terms necessary to evaluate $\tan^{-1}$ and $\tanh^{-1}$ to 32 bits of precision? First consider the value of x for which $\tan^{-1} x = x$ to 32 bits of precision. A theorem of numerical analysis states that for an alternating sum where the absolute values of the terms decrease monotonically, the error is less than the absolute value of the first neglected term. Solving the equation $x^3/3 = 2^{-32}$ for x yields $x = \sqrt[3]{6 \cdot 2^{-11}}$; therefore for $i \ge 11$, $\tan^{-1} 2^{-i} = 2^{-i}$ with 32 bits of precision.

For the higher powers of two, we need to solve the relation $2^{-in}/n = 2^{-32}$ for n for each of the cases $i = 1$ to 10. We do not even attempt the calculation for $i = 0$. The series for $\tan^{-1} 1$ converges very slowly, even after 500 terms the third digit is still changing. Fortunately, we know that the answer is $\pi/4$. Computing the rest is not as much work. The array *terms* has the gory details.

As usual, $\tanh^{-1}$ is more perverse. It is not an alternating sum and does not meet the conditions of the theorem used above. Consider the second neglected term of $\tanh^{-1} 1/2$. It is less than $1/4$ of the first neglected term because the series includes only every other power of two. All of the other neglected terms can have no effect on the 33rd bit. The series for the other arguments, $1/4, 1/8, \ldots$, converges even faster. Therefore, the number of terms calculated for $\tan^{-1}$ works just as well for $\tanh^{-1}$ for 32-bit accuracy.

Before computing the power series, we still need to compute the coefficients, $1/k$, for each term $k = 1, 3, 5, \ldots 27$. We fill the coefficient array *long a[28]* with odd indices by calling the routine *Reciprocal*, which takes two arguments and returns a *long*. The first argument is the integer for the desired reciprocal. The second specifies the desired precision for the fractional part of the result. Reciprocal uses a simple as can be restoring division; it is the algorithm we all learned in grade school for long division. The elements of the array *a* with even indices get *0L* because there are no terms in the power series with even exponents.

Everything is ready to fill the arrays *atan[fractionBits+1]* and *atanh[fractionBits+1]*.

The routine *Poly2* evaluates the power series for the specified number of terms for the specified power of two using Horner's rule. The coefficients come from the array *a*, which we just carefully filled. Horner's rule is the recommended method for evaluating polynomials. A polynomial as in Example 5(e) can be rewritten as in Example 5(f). This simple recursive formula evaluates the polynomial with n multiplications and n additions. We compute the prescaling constants K by using the method explained above; in the program we call these *X0C* and *X0H*, for the circular and hyperbolic constants, respectively. Program output to this point is shown in Listing Two, page 158.

The routines *Circular*, *InvertCircular*, *Hyperbolic*, and *InvertHyperbolic* are the C implementations of the algorithms described above. They all take as arguments the initial values for $[x,y,z]$; they leave their results in the global variables *X*, *Y*, and *Z*. Considering their versatility and the wide range of functions they compute, these routines are compact and elegant!

References

1. R. Nave. "Implementation of Transcendental Functions on a Numerics Processor." *Microprocessing and Microprogramming*, vol. 11, num. 3 – 4, pp. 221 – 225, March – April 1983.

2. A. K. Yuen. "Intel's Floating-Point Processors." *Electro/88 Conference Record*, pp. 48/5/1 – 7, 1988.

3. J. E. Volder. "The Cordic Trigonometric Computing Technique." *IRE Transactions Electronic Computers*, vol. EC – 8, pp. 330 – 334, September 1959.

4. D. S. Cochran. "Algorithms and Accuracy in the HP-35." *Hewlett-Packard Journal*, pp. 10 – 11, June 1972.

5. J. S. Walther. "A Unified Algorithm for Elementary Functions." In *1971 Proceedings of the Joint Spring Computer Conference*, pp. 379 – 385, 1971.

DDJ

(Listings begin on page 157.)

Listing One *(Text begins on page 152.)*

```
/* cordicC.c -- J. Pitts Jarvis, III
 * cordicC.c computes CORDIC constants and exercises the basic algorithms.
 * Represents all numbers in fixed point notation.  1 bit sign,
 * longBits-1-n bit integral part, and n bit fractional part.  n=29 lets us
 * represent numbers in the interval [-4, 4) in 32 bit long.  Two's
 * complement arithmetic is operative here.
 */

#define fractionBits 29
#define longBits 32
#define One     (010000000000>>1)
#define HalfPi (014441766521>>1)

/* cordic algorithm identities for circular functions, starting with [x, y, z]
 *  and then
 *  driving z to 0 gives: [P*(x*cos(z)-y*sin(z)), P*(y*cos(z)+x*sin(z)), 0]
 *  driving y to 0 gives: [P*sqrt(x^2+y^2), 0, z+atan(y/x)]
 *  where K = 1/P = sqrt(1+1)* . . . *sqrt(1+(2^(-2*i)))
 *  special cases which compute interesting functions
 *  sin, cos        [K, 0, a] -> [cos(a), sin(a), 0]
 *  atan            [1, a, 0] -> [sqrt(1+a^2)/K, 0, atan(a)]
 *                  [x, y, 0] -> [sqrt(x^2+y^2)/K, 0, atan(y/x)]
 * for hyperbolic functions, starting with [x, y, z] and then
 * driving z to 0 gives: [P*(x*cosh(z)+y*sinh(z)), P*(y*cosh(z)+x*sinh(z)), 0]
 * driving y to 0 gives: [P*sqrt(x^2-y^2), 0, z+atanh(y/x)]
 * where K = 1/P = sqrt(1-(1/2)^2)* . . . *sqrt(1-(2^(-2*i)))
 *  sinh, cosh      [K, 0, a] -> [cosh(a), sinh(a), 0]
 *  exponential     [K, K, a] -> [e^a, e^a, 0]
 *  atanh           [1, a, 0] -> [sqrt(1-a^2)/K, 0, atanh(a)]
 *                  [x, y, 0] -> [sqrt(x^2-y^2)/K, 0, atanh(y/x)]
 *  ln              [a+1, a-1, 0] -> [2*sqrt(a)/K, 0, ln(a)/2]
 *  sqrt            [a+(K/2)^2, a-(K/2)^2, 0] -> [sqrt(a), 0, ln(a*(2/K)^2)/2]
 *  sqrt, ln        [a+(K/2)^2, a-(K/2)^2, -ln(K/2)] -> [sqrt(a), 0, ln(a)/2]
 * for linear functions, starting with [x, y, z] and then
 *  driving z to 0 gives: [x, y+x*z, 0]
 *  driving y to 0 gives: [x, 0, z+y/x]
 */

long X0C, X0H, X0R; /* seed for circular, hyperbolic, and square root */
long OneOverE, E;   /* the base of natural logarithms */
long HalfLnX0R;     /* constant used in simultanous sqrt, ln computation */

/* compute atan(x) and atanh(x) using infinite series
 *    atan(x) =  x - x^3/3 + x^5/5 - x^7/7 + . . . for x^2 < 1
 *    atanh(x) = x + x^3/3 + x^5/5 + x^7/7 + . . . for x^2 < 1
 * To calcuate these functions to 32 bits of precision, pick
 * terms[i] s.t. ((2^-i)^(terms[i]))/(terms[i]) < 2^-32
 * For x <= 2^(-11), atan(x) = atanh(x) = x with 32 bits of accuracy  */
unsigned terms[11]= {0, 27, 14, 9, 7, 5, 4, 4, 3, 3, 3};
static long a[28], atan[fractionBits+1], atanh[fractionBits+1], X, Y, Z;
#include <stdio.h>      /* putchar is a marco for some */

/* Delta is inefficient but pedagogical */
#define Delta(n, Z) (Z>=0) ? (n) : -(n)
#define abs(n) (n>=0) ? (n) : -(n)

/* Reciprocal, calculate reciprocol of n to k bits of precision
 *  a and r form integer and fractional parts of the dividend respectively  */
long
Reciprocal(n, k) unsigned n, k;
{
  unsigned i, a= 1; long r= 0;
  for (i= 0; i<=k; ++i) {r += r; if (a>=n) {r += 1; a -= n;}; a += a;}
  return(a>=n? r+1 : r); /* round result */
}

/* ScaledReciprocal, n comes in funny fixed point fraction representation */
long
ScaledReciprocal(n, k) long n; unsigned k;
{
  long a, r=0; unsigned i;
  a= 1L<<k;
  for (i=0; i<=k; ++i) {r += r; if (a>=n) {r += 1; a -= n;}; a += a;};
  return(a>=n? r+1 : r); /* round result */
}

/* Poly2 calculates polynomial where the variable is an integral power of 2,
 *      log is the power of 2 of the variable
 *      n is the order of the polynomial
 *      coefficients are in the array a[]   */
long
Poly2(log, n) int log; unsigned n;
{
  long r=0; int i;
  for (i=n; i>=0; --i) r= (log<0? r>>-log : r<<log)+a[i];
  return(r);
}
WriteFraction(n) long n;
{
  unsigned short i, low, digit; unsigned long k;
  putchar(n < 0 ? '-' : ' '); n = abs(n);
  putchar((n>>fractionBits) + '0'); putchar('.');
  low = k = n << (longBits-fractionBits); /* align octal point at left */
  k   >>=  4;   /* shift to make room for a decimal digit */
  for (i=1; i<=8; ++i)
  {
    digit = (k *= 10L) >> (longBits-4);
    low = (low & 0xf) * 10;
    k += ((unsigned long) (low>>4)) - ((unsigned long) digit << (longBits-4));
    putchar(digit+'0');
  }
}
WriteRegisters()
{
  printf("  X: "); WriteVarious(X);
  printf("  Y: "); WriteVarious(Y);
  printf("  Z: "); WriteVarious(Z);
}
WriteVarious(n) long n;
{
  WriteFraction(n); printf("  0x%08lx 0%011lo\n", n, n);
}
Circular(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=0; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atan[i];
    X -= Delta(y, Z);
    Y += Delta(x, Z);
    Z -= Delta(z, Z);
  }
}
InvertCircular(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=0; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atan[i];
    X -= Delta(y, -Y);
    Z -= Delta(z, -Y);
    Y += Delta(x, -Y);
  }
```

(continued on page 158)

Listing One *(Listing continued, text begins on page 152.)*

```
}
Hyperbolic(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=1; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atanh[i];
    X += Delta(y, Z);
    Y += Delta(x, Z);
    Z -= Delta(z, Z);
    if ((i==4)||(i==13))
    {
      x= X>>i; y= Y>>i; z= atanh[i];
      X  +=  Delta(y, Z);
      Y += Delta(x, Z);
      Z  -=  Delta(z, Z);
    }
  }
}
InvertHyperbolic(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z;
  for (i=1; i<=fractionBits; ++i)
  {
    x= X>>i; y= Y>>i; z= atanh[i];
    X += Delta(y, -Y);
    Z -= Delta(z, -Y);
    Y += Delta(x, -Y);
    if ((i==4)||(i==13))
    {
      x= X>>i; y= Y>>i; z= atanh[i];
      X += Delta(y, -Y);
      Z -= Delta(z, -Y);
      Y += Delta(x, -Y);
    }
  }
}
Linear(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z; z= One;
  for (i=1; i<=fractionBits; ++i)
  {
    x  >>= 1; z  >>= 1; Y += Delta(x, Z); Z -= Delta(z, Z);
  }
}
InvertLinear(x, y, z) long x, y, z;
{
  int i;
  X = x; Y = y; Z = z; z= One;
  for (i=1; i<=fractionBits; ++i)
  {
     Z -= Delta(z  >>= 1, -Y); Y += Delta(x  >>= 1, -Y);
  }
}

/*********************************************************/
main()
{
  int i; long r;
  /*system("date");*//* time stamp the log for UNIX systems */
  for (i=0; i<=13; ++i)
  {
    a[2*i]= 0; a[2*i+1]= Reciprocal(2*i+1, fractionBits);
  }
  for (i=0; i<=10; ++i) atanh[i]= Poly2(-i, terms[i]);
  atan[0]= HalfPi/2; /* atan(2^0)= pi/4 */
  for (i=1; i<=7; ++i) a[4*i-1]= -a[4*i-1];
  for (i=1; i<=10; ++i) atan[i]= Poly2(-i, terms[i]);
  for (i=11; i<=fractionBits; ++i) atan[i]= atanh[i]= 1L<<(fractionBits-i);
  printf("\natanh(2^-n)\n");
  for (i=1; i<=10; ++i){printf("%2d ", i); WriteVarious(atanh[i]);}
  r= 0;
  for (i=1; i<=fractionBits; ++i)
     r +=  atanh[i];
  r +=  atanh[4]+atanh[13];
  printf("radius of convergence"); WriteFraction(r);
  printf("\n\natan(2^-n)\n");
  for (i=0; i<=10; ++i){printf("%2d ", i); WriteVarious(atan[i]);}
  r= 0; for (i=0; i<=fractionBits; ++i) r +=  atan[i];
  printf("radius of convergence"); WriteFraction(r);

  /* all the results reported in the printfs are calculated with my HP-41C */
  printf("\n\n--------------------circular functions--------------------\n");
    printf("Grinding on [1, 0, 0]\n");
    Circular(One, 0L, 0L); WriteRegisters();
    printf("\n  K: "); WriteVarious(X0C= ScaledReciprocal(X, fractionBits));
    printf("\nGrinding on [K, 0, 0]\n");
    Circular(X0C, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/6] -> [0.86602540, 0.50000000, 0]\n");
    Circular(X0C, 0L, HalfPi/3L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/4] -> [0.70710678, 0.70710678, 0]\n");
    Circular(X0C, 0L, HalfPi/2L); WriteRegisters();
    printf("\nGrinding on [K, 0, pi/3] -> [0.50000000, 0.86602540, 0]\n");
    Circular(X0C, 0L, 2L*(HalfPi/3L)); WriteRegisters();
  printf("\n------Inverse functions------\n");
    printf("Grinding on [1, 0, 0]\n");
    InvertCircular(One, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [1, 1/2, 0] -> [1.84113394, 0, 0.46364761]\n");
    InvertCircular(One, One/2L, 0L); WriteRegisters();
    printf("\nGrinding on [2, 1, 0] -> [3.68226788, 0, 0.46364761]\n");
    InvertCircular(One*2L, One, 0L); WriteRegisters();
    printf("\nGrinding on [1, 5/8, 0] -> [1.94193815, 0, 0.55859932]\n");
    InvertCircular(One, 5L*(One/8L), 0L); WriteRegisters();
    printf("\nGrinding on [1, 1, 0] -> [2.32887069, 0, 0.78539816]\n");
    InvertCircular(One, One, 0L); WriteRegisters();
  printf("\n--------------------hyperbolic functions--------------------\n");
    printf("Grinding on [1, 0, 0]\n");
    Hyperbolic(One, 0L, 0L); WriteRegisters();
    printf("\n  K: "); WriteVarious(X0H= ScaledReciprocal(X, fractionBits));
    printf("  R: "); X0R= X0H>>1; Linear(X0R, 0L, X0R); WriteVarious(X0R= Y);
    printf("\nGrinding on [K, 0, 0]\n");
    Hyperbolic(X0H, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [K, 0, 1] -> [1.54308064, 1.17520119, 0]\n");
    Hyperbolic(X0H, 0L, One); WriteRegisters();
    printf("\nGrinding on [K, K, -1] -> [0.36787944, 0.36787944, 0]\n");
    Hyperbolic(X0H, X0H, -One); WriteRegisters();
    OneOverE = X; /* save value ln(1/e) = -1 */
    printf("\nGrinding on [K, K, 1] -> [2.71828183, 2.71828183, 0]\n");
    Hyperbolic(X0H, X0H, One); WriteRegisters();
    E = X; /* save value ln(e) = 1 */
  printf("\n------Inverse functions------\n");
    printf("Grinding on [1, 0, 0]\n");
    InvertHyperbolic(One, 0L, 0L); WriteRegisters();
    printf("\nGrinding on [1/e + 1, 1/e - 1, 0] -> [1.00460806, 0,
                                                    -0.50000000]\n");
    InvertHyperbolic(OneOverE+One,OneOverE-One, 0L); WriteRegisters();
    printf("\nGrinding on [e + 1, e - 1, 0] -> [2.73080784, 0, 0.50000000]\n");
    InvertHyperbolic(E+One, E-One, 0L); WriteRegisters();
    printf("\nGrinding on (1/2)*ln(3) -> [0.71720703, 0, 0.54930614]\n");
    InvertHyperbolic(One, One/2L, 0L); WriteRegisters();
    printf("\nGrinding on [3/2, -1/2, 0] -> [1.17119417, 0, -0.34657359]\n");
    InvertHyperbolic(One+(One/2L), -(One/2L), 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1/2) -> [0.70710678, 0, 0.15802389]\n");
    InvertHyperbolic(One/2L+X0R, One/2L-X0R, 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1) -> [1.00000000, 0, 0.50449748]\n");
    InvertHyperbolic(One+X0R, One-X0R, 0L); WriteRegisters();
    HalfLnX0R = Z;
    printf("\nGrinding on sqrt(2) -> [1.41421356, 0, 0.85117107]\n");
    InvertHyperbolic(One*2L+X0R, One*2L-X0R, 0L); WriteRegisters();
    printf("\nGrinding on sqrt(1/2), ln(1/2)/2 -> [0.70710678, 0,
                                                    -0.34657359]\n");
    InvertHyperbolic(One/2L+X0R, One/2L-X0R, -HalfLnX0R); WriteRegisters();
    printf("\nGrinding on sqrt(3)/2, ln(3/4)/2 -> [0.86602540, 0,
                                                    -0.14384104]\n");
    InvertHyperbolic((3L*One/4L)+X0R, (3L*One/4L)-X0R, -HalfLnX0R);
    WriteRegisters();
    printf("\nGrinding on sqrt(2), ln(2)/2 -> [1.41421356, 0, 0.34657359]\n");
    InvertHyperbolic(One*2L+X0R, One*2L-X0R, -HalfLnX0R);
    WriteRegisters();
    exit(0);
}
```

End Listing One

Listing Two

```
atanh (2^-n)

1   0.54930614   0x1193ea7a 002144765172
2   0.25541281   0x082c577d 001013053575
3   0.12565721   0x04056247 000401261107
4   0.06258157   0x0200ab11 000200125421
5   0.03126017   0x01001558 000100012530
6   0.01562627   0x008002aa 000040001252
7   0.00781265   0x00400055 000020000125
8   0.00390626   0x0020000a 000010000012
9   0.00195312   0x00100001 000004000001
10  0.00097656   0x00080000 000002000000

radius of convergence 1.11817300

atan (26-n)

0   0.78539816   0x1921fb54 003110375524
1   0.46364760   0x0ed63382 001665431602
2   0.24497866   0x07d6dd7e 000765556576
3   0.12435499   0x03fab753 000376533523
4   0.06241880   0x01ff55bb 000177652673
5   0.03123983   0x00ffeaad 000077765255
6   0.01562372   0x007ffd55 000037776525
7   0.00781233   0x003fffaa 000017777652
8   0.00390622   0x001ffff5 000007777765
9   0.00195312   0x000ffffe 000003777776
10  0.00097656   0x0007ffff 000001777777

radius of convergence 1.74328660

------- circular functions ------------------


K: 0.60725293       0x136e9db3     002333516663


------- hyperbolic functions ----------------


K: 1.20749708       0x26a3d0ed     004650750355

R: 0.36451229       0xbaa15b1      001352412661
```

End Listings