# CORDIC Algorithm Simulation Code

```
/*
 * CORDIC algorithms.
 * The original code was published in Docter Dobbs Journal issue ddj9010.
 * The ddj version can be obtained using FTP from SIMTEL and other places.
 *
 * Converted to ANSI-C (with prototypes) by P. Knoppers, 13-Apr-1992.
 *
 * The main advantage of the CORDIC algorithms is that all commonly used math
 * functions ([a]sin[h] [a]cos[h] [a]tan[h] atah[h](y/x) ln exp sqrt) are
 * implemented using only shift, add, subtract and compare. All values are
 * treated as integers. Actually they are fixed point floating point values.
 * The position of the fixed point is a compile time constant (fractionBits).
 * I don't believe that this can be easily fixed...
 *
 * Some initialization of internal tables and constants is necessary before
 * all functions can be used. The constant "HalfPi" must be determined before
 * compile time, all others are computed during run-time - see main() below.
 *
 * Of course, any serious implementation of these functions should probably
 * have _all_ constants determined sometime before run-time and most
 * functions might be written in assembler.
 *
 *
 * The layout of the code is adapted to my personal preferences.
 *
 * PK.
 */
/* Prototypes: (these should be moved to a separate include file) */

void WriteVarious (long n);
void WriteFraction (long n);
long Reciprocal (unsigned n, unsigned k);
long Poly2 (int log, unsigned n);
void Circular (long x, long y, long z);
void WriteRegisters (void);
long ScaledReciprocal (long n, unsigned k);
void InvertCircular (long x, long y, long z);
void Hyperbolic (long x, long y, long z);
void Linear (long x, long y, long z);
void InvertHyperbolic (long x, long y, long z);

/*
 * _IMPLEMENTING CORDIC ALGORITHMS_
 * by Pitts Jarvis
 *
 *
 * [LISTING ONE]
 */

/*
 * 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             (010000000000L>>1)
#define HalfPi          (014441766521L>>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
 */

static unsigned terms[11] = {0, 27, 14, 9, 7, 5, 4, 4, 3, 3, 3};
static long a[28];
static long atan[fractionBits + 1];
static long atanh[fractionBits + 1];
static long X;
static long Y;
static long 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 (unsigned n, unsigned k)
{
    unsigned i;
    unsigned 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 (long n, unsigned k)
{
    long a;
    long r = 0L;
    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 (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);
```

```
}

void WriteFraction (long n)
{
    unsigned short i;
    unsigned short low;
    unsigned short 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');
    }
}

void WriteRegisters (void)
{
    printf ("  X: ");
    WriteVarious (X);
    printf ("  Y: ");
    WriteVarious (Y);
    printf ("  Z: ");
    WriteVarious (Z);
}

void WriteVarious (long n)
{
    WriteFraction (n);
    printf ("  0x%08lx 0%011lo\n", n, n);
}

void Circular (long x, long y, long 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);
    }
}

void InvertCircular (long x, long y, long 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);
    }
}

void Hyperbolic (long x, long y, long 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);
        }
    }
}

void InvertHyperbolic (long x, long y, long 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);
        }
    }
}

void Linear (long x, long y, long 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);
    }
}

void InvertLinear (long x, long y, long 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);
    }
}

/*********************************************************/

int main (int argc, char *argv[])
{
    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 ();
    return (0);
}


/*
[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 (2^-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
*/
```

---

## Comments from Peter Knoppers

```
Good idea setting up a page about Cordic algorithms!

Maybe you would like to add a page with the program that I have had
lying around for years now. It nicely demostrates Cordic algorithms.
Requires ANSI C-compiler.

Greetings from Delft, the Netherlands,
```


```
 Delft University of Technology, The Netherlands.    Phone +31-(0)15-2144886
 Email: knop@duteca.et.tudelft.nl  WWW: http://cardit.et.tudelft.nl/~knop/
```


---