Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions

By Andrea Vitali

### Main components

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
</table>

### Purpose and benefits

This design tip explains how to compute trigonometric and hyperbolic functions using the coordinate rotation digital computer algorithm (CORDIC).

- This algorithm has very low complexity, using only shifts and adds. No floating point math. No FPU/DSP needed, therefore it is suited for cores such as the Cortex-M0.

- This algorithm has a very low memory footprint, using only a small look up table (LUT), with as many integer entries as the number of precision bits required. The ARM CMSIS library occupies 220kB, while CORDIC may be less than 1kB.

- This algorithm is fast, performing, on average, a number of iterations equal to the number of precision bits required. With respect to the single precision FPU that takes on average a constant time (T), the ARM CMSIS can be faster (T to T/3) and the CORDIC can be slower (T to 3T). Software emulation of single/double precision floating point math is the slowest solution (10T to 30T).

- The following trigonometric functions can be computed: sin(), cos(), atan(), atan2(). The following hyperbolic functions can be computed: sinh(), cosh(), atanh(). Other functions that are computed as a byproduct: sqrt(), exp(), ln(). The CORDIC always computes functions in pairs (see table below).

A utility is provided, in C source code format, to automatically generate the look up table (LUT) used by the CORDIC algorithm for all three coordinates system: circular, linear and hyperbolic.

A reference fixed-point C source code is provided for all CORDIC modes: rotation and vectoring. A specialized code is also provided for the case of trigonometric functions.

In the companion Design Tip, DT0087 test code is provided to verify performance and error bounds.
Description

The CORDIC algorithm can be seen as a sequence of micro rotations (see Figure 1), where the vector XY is rotated by an angle A. Remembering that \(\tan(A) = \sin(A)/\cos(A)\), the formula for the single micro rotation is the following:

\[
X_{n+1} = \cos(A) X_n - \sin(A) Y_n \\
Y_{n+1} = \sin(A) X_n + \cos(A) Y_n
\]

The rotation angle is chosen so that the \(\tan(A)\) coefficient is a power of two, therefore the multiplication is reduced to a bit shift. If the components are scaled by \(F = 1/\cos(A)\), which is the CORDIC gain, the formula for the rotation is reduced to only bit shifts and additions:

\[
X_{n+1} F_n = [ X_n - Y_n / 2^n ] \\
Y_{n+1} F_n = [ X_n / 2^n + Y_n ]
\]

The elementary rotation angle is \(A_n = \arctan(1/2^n)\). The corresponding scaling factor is \(F_n = 1/\cos(A_n) = \sqrt{1+1/2^{2n}}\).

The unified CORDIC algorithm

The unified CORDIC algorithm uses three registers: X and Y for the vector, and Z for angle. Only shifts and adds are done. The scaling factor is compensated at the end of the loop.

\[
X_{n+1} F_n = [ X_n - MS Y_n / 2^n ] \\
Y_{n+1} F_n = [ SX_n / 2^n + Y_n ] \\
Z_{n+1} = Z_n - S a_n
\]

The factor M, the elementary angle \(A_n\), and the corresponding scaling \(F_n\) are determined by the CORDIC coordinate system in use (see Figure 1):

- Circular: \(A_n = \arctan(1/2^n)\), \(n\) from 0, \(M = +1\), \(F_n = \sqrt{1+1/2^{2n}}\)
- Linear: \(A_n = 1/2^n\), \(n\) from 0, \(M = 0\), \(F_n = \sqrt{1}\)
- Hyperbolic: \(A_n = \arctanh(1/2^n)\), \(n\) from 1, \(M = -1\), \(F_n = \sqrt{1-1/2^{2n}}\)

The factor S is determined by the CORDIC operation mode:

- Rotation: the angle Z is driven to 0; \(S = +1\) if \(Z \geq 0\), \(S = -1\) if \(Z < 0\).
- Vectoring: the Y component is driven to 0; \(S = +1\) if \(Y \leq 0\), \(S = -1\) if \(Y > 0\).

The angle in register Z must be less than the convergence angle, which is the sum of the angles \(A_n\) at each iteration. For the hyperbolic coordinate system, the following iterations must be repeated to obtain convergence: 4, 13, 40, ... \(k\), ... \(3k+1\).

- Circular: \(A = \sum A_n = 1.7432866\) radians (99.9deg), \(n = 0, 1, 2, 3, 4, 5, \ldots\) \(N\)
- Linear: \(A = \sum A_n = 2\), \(n = 0, 1, 2, 3, 4, 5, \ldots\) \(N\)
- Hyperbolic: \(A = \sum A_n = 1.1181730\) radians (64deg), \(n = 1, 2, 3, 4, 5, \ldots\) \(N\)
As only shifts and adds are done at each iteration, at the end of the loop the final vector components X and Y are scaled by the product of the scaling factors $F_n$.

- **Circular**: $F = \prod F_n = 1.64676025812107$, $1/F = 0.60725293500881$
- **Hyperbolic**: $F = \prod F_n = 0.82978162013890$, $1/F = 1.20513635844646$

**Functions computed by CORDIC**

The CORDIC algorithm always computes two functions simultaneously.

In the circular coordinate system: the $\sin()$ & $\cos()$ pair is frequently used in I/Q demodulators and direct digital frequency synthesizers DDFS. The $\atan()$ & $\sqrt{}$ are conveniently used in PM/AM demodulators.

The function $\atan2(y,x)$ can be computed based on $\atan(y/x)$:

- If $x\geq 0$, $\atan2(y,x) = \atan(y/x)$
- If $x<0$ and $y\geq 0$, $\atan2(y,x) = \atan(-y/x) + \pi$
- If $x<0$ and $y<0$, $\atan2(y,x) = \atan(-y/x) - \pi$
### Circular coordinate system. Rotation mode: Z is driven to 0.  \( |Z| < 1.74 \) (99.9deg),  \( F \approx 1.64 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( a )</td>
<td>( x F \left[ x \cos(a) - y \sin(a) \right] )</td>
<td>( y F \left[ x \sin(a) + y \cos(a) \right] )</td>
<td>( z = 0 )</td>
</tr>
<tr>
<td>( m ),  ( 0 ),  ( a )</td>
<td>( x F m \cos(a) )</td>
<td>( y F m \sin(a) )</td>
<td>( z = 0 )</td>
</tr>
<tr>
<td>( 1/F ),  ( 0 ),  ( a )</td>
<td>( x \cos(a) )</td>
<td>( y \sin(a) )</td>
<td>( z = 0 )</td>
</tr>
</tbody>
</table>

### Circular coordinate system. Vectoring mode: Y is driven to 0.  \( |Z| < 1.74 \) (99.9deg),  \( F \approx 1.64 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( z )</td>
<td>( x F \sqrt{x^2 + y^2} )</td>
<td>( y = 0 )</td>
<td>( z = z + \tan(y/x) )</td>
</tr>
<tr>
<td>( 1 ),  ( a ),  ( 0 )</td>
<td>( x F \sqrt{1 + a^2} )</td>
<td>( y = 0 )</td>
<td>( z = \tan(a) )</td>
</tr>
<tr>
<td>( a ),  ( 1 ),  ( 0 )</td>
<td>( x F \sqrt{a^2 + 1} )</td>
<td>( y = 0 )</td>
<td>( z = \cot(a) )</td>
</tr>
</tbody>
</table>

The linear coordinate system allows multiply-and-accumulate operations. Also, it can be used to perform division operations.

### Linear coordinate system. Rotation mode: Z is driven to 0.  \( |Z| < 2 \),  \( F = 1 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( z )</td>
<td>( x )</td>
<td>( y + x z )</td>
<td>( z = 0 )</td>
</tr>
</tbody>
</table>

### Linear coordinate system. Vectoring mode: Y is driven to 0.  \( |Z| < 2 \),  \( F = 1 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( z )</td>
<td>( x )</td>
<td>( y = 0 )</td>
<td>( z = z + y / x )</td>
</tr>
</tbody>
</table>

Finally, the hyperbolic coordinate system is used to compute the \( \exp() \) and \( \ln() \) functions. Remember that \( \exp(a) = \sinh(a) + \cosh(a) \). Also remember that \( \tanh(y/x) = \frac{1}{2} \ln((x+y)/(x-y)) \) and therefore \( \tanh(a) = \frac{1}{2} \ln((1+a)/(1-a)) \).

### Hyperbolic coordinate system. Rotation mode: Z is driven to 0.  \( |Z| < 1.11 \) (64deg),  \( F \approx 0.82 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( a )</td>
<td>( x F \left[ x \cosh(a) + y \sinh(a) \right] )</td>
<td>( y F \left[ x \sinh(a) + y \cosh(a) \right] )</td>
<td>( z = 0 )</td>
</tr>
<tr>
<td>( 1/F ),  ( 0 ),  ( a )</td>
<td>( x \cosh(a) )</td>
<td>( y \sinh(a) )</td>
<td>( z = 0 )</td>
</tr>
<tr>
<td>( b/F ),  ( b/F ),  ( a )</td>
<td>( x b \exp(a) )</td>
<td>( y b \exp(a) )</td>
<td>( z = 0 )</td>
</tr>
</tbody>
</table>

### Circular coordinate system. Vectoring mode: Y is driven to 0.  \( |Z| < 1.11 \) (64deg),  \( F \approx 0.82 \)

<table>
<thead>
<tr>
<th>( x ),  ( y ),  ( z )</th>
<th>( x_N )</th>
<th>( y_N )</th>
<th>( z_N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ),  ( y ),  ( a )</td>
<td>( x F \sqrt{x^2 - y^2} )</td>
<td>( y = 0 )</td>
<td>( z = z + \tanh(y/x) )</td>
</tr>
<tr>
<td>( 1 ),  ( a ),  ( 0 )</td>
<td>( x F \sqrt{1 - a^2} )</td>
<td>( y = 0 )</td>
<td>( z = \tanh(a) )</td>
</tr>
<tr>
<td>( a ),  ( 1 ),  ( 0 )</td>
<td>( x F \sqrt{a^2 - 1} )</td>
<td>( y = 0 )</td>
<td>( z = \coth(a) )</td>
</tr>
<tr>
<td>( a + b ),  ( a - b ),  ( 0 )</td>
<td>( x F (a b) )</td>
<td>( y = 0 )</td>
<td>( z = \ln(a/b) / 2 )</td>
</tr>
<tr>
<td>( a + 1 ),  ( a - 1 ),  ( 0 )</td>
<td>( x F \sqrt{a} )</td>
<td>( y = 0 )</td>
<td>( z = \ln(a) / 2 )</td>
</tr>
<tr>
<td>( a + 1/4 ),  ( a - 1/4 ),  ( z )</td>
<td>( x F \sqrt{a} )</td>
<td>( y = 0 )</td>
<td>( z = \ln(4 a) / 2 )</td>
</tr>
</tbody>
</table>
CORDIC fixed-point C implementation

Look Up Table generation code

The utility generates the file CORDICtable.c, which is to be included in CORDIC.c. The name is modified with the suffix _LIN or _HYPER when circular or hyperbolic coordinates are selected. The code below is formatted for compactness, not for readability.

```c
#include <stdio.h>
#include <math.h>

#define M_PI 3.1415926536897932384626

int main(int argc, char *argv[]) {
    FILE *f; char tname[50], cname[10];
    int n,n2,mp2,niter,bits,t;
    double F, A, mul, tmul;

    // CORDIC gain, convergence angle, multiplication factor
    printf("@fCircular, 1)linear, 2)hyperbolic? "); scanf("%d",&t); switch(t) {
        case 0: printf("%s",""); break;
        case 1: printf("%s","_LIN"); break;
        case 2: printf("%s","_HYPER"); break;
    }
    sprintf(tname,"CORDICtable%s.c",cname);

    if(NULL==(f=fopen(tname,"wt"))) { printf("cannot write to %s\n",tname); return 0; }

    printf("number of bits for mantissa (e.g. 38)? "); scanf("%d",&bits);
    printf("0) mul factor is 2^n (easier output scaling), on\n" "1) 2pi is 2^n (easier implementation)\n? "); scanf("%s",&mp2);
    printf("suggested multiplication factor\n");
    if(mp2==0) { tmul=(double)(1<<(bits-3)); printf("2^%d = %f\n", bits-3,tmul); }
    else { tmul=(double)(1<<(bits-2))/M_PI; printf("2^%d/pi = %f\n",bits-2,tmul); }

    printf("multiplication factor (0 for suggested)? "); scanf("%f",&mul);
    if(mul<0.1) { mul=tmul; printf("%f\n",mul); }
    else mp2=1; // custom factor

    switch(t) {
        case 0: for(n=0; n<bits;n++) {
            if((int)round(atan(pow(2.0,(double)(-n)))*mul)==0) break; break;
        }
        case 1: for(n=0; n<bits;n++) {
            if((int)round(atan(pow(2.0,(double)(-n)))*mul)==0) break; break;
        }
        case 2: for(n=1,n2=4,nbits; ) {
            if((int)round(atanh(pow(2.0,(double)(-n)))*mul)==0) break; break;
            if((n+n2) n2=3*n+1; else n++; }
        }
    }

    printf("iterations (up to %d)\n",n); scanf("%d",&niter);

    F=1.0; A=0.0; switch(t) {
        case 0: for(n=0; n<nbits;n++) {
            F=F*sqrt(1+pow(2.0,(double)(-n)));
            A+=atan(pow(2.0,(double)(-n)))*mul;
        }
        case 1: for(n=0; n<nbits;n++) {
            F=F*sqrt(1+pow(2.0,(double)(-n)));
            A+=atan(pow(2.0,(double)(-n)))*mul;
        }
        case 2: for(n=1,n2=4,nbits; ) {
            F=F*sqrt(1+pow(2.0,(double)(-n)));
            A+=atan(pow(2.0,(double)(-n)))*mul;
            if((n+n2) n2=3*n+1; else n++; }
        }
    }

    fprintf(f,"//CORDIC\n",cname,bits,niter);
    fprintf(f,"// 1.0 = %f multiplication factor\n",mul);

    switch(t) {
        case 0: fprintf(f,"// A = %f convergence angle\n",A);
            fprintf(f,"// F = %f gain limit (limit is 0.82978162013890)\n",F);
            fprintf(f,"// 1/F = %f inverse gain\n",1.0/F);
        case 1: fprintf(f,"// A = %f convergence angle (limit is 2)\n",A);
            fprintf(f,"// F = %f gain limit (limit is 0.607252935008881)\n",F);
            fprintf(f,"// 1/F = %f inverse gain\n",1.0/F);
        case 2: fprintf(f,"// A = %f convergence angle\n",A);
            fprintf(f,"// F = %f gain limit (limit is 1.64676025812107)\n",F);
            fprintf(f,"// 1/F = %f inverse gain\n",1.0/F);
        }
    }

    fprintf(f,"// pi = %f (3.1415926536897932384626)\n",M_PI);
    fprintf(f,"\n");
    define CORDICS_A %f // CORDIC convergence angle A\n",cname,A);
    fprintf(f,"//CORDIC gain F\n",F);
    fprintf(f,"//CORDIC inverse gain 1/F\n",F);
    fprintf(f,"\n");
    define CORDICS_F 0x588X // CORDIC gain F\n",F);
    fprintf(f,"\n");
}
```

#define CORDIC%s_HALFPI   0x%08X
    n
cname,(int)round(mul*(M_PI/2.0)));
#define CORDIC%s_PI       0x%08X
    n
cname,(int)round(mul*(M_PI)));
#define CORDIC%s_TWOPI    0x%08X
    n
cname,(int)round(mul*(2.0*M_PI)));
#define CORDIC%s_MUL      %f // CORDIC multiplication factor M
    n
cname,mul);
switch (mp2) {
case 0:  fprintf(f,
    n
    " = 2^%d
    n",
    n
    bits
    -
    3);
    break
    n
case 1:
    fprintf(f,
    n
    " = 2^%d/pi
    n",
    n
    bits
    -
    2);
    break
    n
default: fprintf(f,
    n
    "n";
    break
    n
switch (t) {
case 0: fprintf(f,
    n
    "0x%08X
    n",
    n
    (int)
    round( atan(pow(2.0,
    n
    "(double)(
    n
    "-(
    n
    ")*mul));
    break
    n
case 1: fprintf(f,
    n
    "0x%08X
    n",
    n
    (int)
    round(     (pow(2.0,
    n
    "(double)(
    n
    "-(
    n
    ")*mul));
    break
    n
case 2: n=n==0?1:n;
    fprintf(f,
    n
    "0x%08X
    n",
    n
    (int)
    round(atanh(pow(2.0,
    n
    "(double)(
    n
    "-(
    n
    ")*mul));
    break
    n
    if (n<(niter
    n
    -
    1)) fprintf(f,
    n
    " ,
    n
    "
    n
    else
    fprintf(f,
    n
    "
    n
    }
}
#define CORDIC%s_MAXITER  %d
    n
cname,niter);
#include "CORDICtable.c"
// z less than convergence angle (limit is 1.7432866 = 99.9deg) multiplied by M
void CORDIC_rotation_Zto0(int x, int y, int z, int *xx, int *yy) {
    int k, tx;
    for (k=0; k<CORDIC_MAXITER; k++) {
        tx = x;
        if (z>=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
        else { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; }
        *xx = x; // *x*cos(z)-y*sin(z) multiplied by M and gain F
        *yy = y; // *x*sin(z)+y*cos(z) multiplied by M and gain F
    }
}
void CORDIC_vectoring_Yto0(int x, int y, int z, int *xx, int *zz) {
    int k, tx;
    for (k=0; k<CORDIC_MAXITER; k++) {
        tx = x;
        if (y<=0) { x += (y>>k); y -= (tx>>k); z -= CORDIC_ZTBL[k]; }
        else { x -= (y>>k); y += (tx>>k); z += CORDIC_ZTBL[k]; }
        *xx = x; // sqrt(x^2+y^2) multiplied by gain F
        *zz = z; // z+atan2(y,x) multiplied by M
    }
}
#include "CORDICtable_LIN.c"
// z less than convergence angle (limit is 2) multiplied by M
void CORDIC_LIN_rotation_Zto0(int x, int y, int z, int *xx, int *yy) {
    int k, tx;
    for (k=0; k<CORDIC_LIN_MAXITER; k++) {
        tx = x;
        if (z>=0) { y += (tx>>k); z -= CORDIC_LIN_ZTBL[k]; }
        else { y -= (tx>>k); z += CORDIC_LIN_ZTBL[k]; }
        *xx = x; // x multiplied by M (gain F=1)
        *yy = y; // y+x*z multiplied by M (gain F=1)
    }
}
void CORDIC_LIN_vectoring_Yto0(int x, int y, int z, int *xx, int *zz) {
    int k, tx;
    for (k=0; k<CORDIC_LIN_MAXITER; k++) {
        tx = x;
        if (y<=0) { y += (tx>>k); z -= CORDIC_LIN_ZTBL[k]; }
        else { y -= (tx>>k); z += CORDIC_LIN_ZTBL[k]; }
        *xx = x; // x as is (gain F=1)
        *zz = z; // z+y/x multiplied by M
    }
CORDIC code for hyperbolic coordinates

```c
#include "CORDICtable_HYPER.c"

// z less than convergence angle (limit is 1.1181730 = 64.0deg) multiplied by M
void CORDIC_HYPER_rotation_Zto0(int x, int y, int z, int *xx, int *yy)
{
    int k, k2, tx;
    for (k=1, k2=4; k<CORDIC_HYPER_MAXITER;)
    {
        tx = x;
        if (z>=0) { x += (y>>k); y += (tx>>k); z = CORDIC_HYPER_ZTBL[k]; }
        else { x = (y>>k); y = (tx>>k); z += CORDIC_HYPER_ZTBL[k]; }
        if (k==k2) k2=k*3+1; else k++;
    }
    *xx = x; // x*cosh(z)+y*sinh(z) multiplied by M and gain F
    *yy = y; // x*sinh(z)+y*cosh(z) multiplied by M and gain F
}

void CORDIC_HYPER_vectoring_Yto0(int x, int y, int z, int *xx, int *zz)
{
    int k, k2, tx;
    for (k=1, k2=4; k<CORDIC_HYPER_MAXITER;)
    {
        tx = x;
        if (y<=0) { x += (y>>k); y += (tx>>k); z = CORDIC_HYPER_ZTBL[k]; }
        else { x = (y>>k); y = (tx>>k); z += CORDIC_HYPER_ZTBL[k]; }
        if (k==k2) k2=k*3+1; else k++;}
    *xx = x; // sqrt(x^2+y^2) multiplied by gain F
    *zz = z; // z+atan2(y,x) multiplied by M
}
```

CORDIC code circular coordinates, specialized for trigonometric functions

```c
#include "CORDICtable.c"

// angle is radians multiplied by CORDIC multiplication factor M
// modulus can be set to CORDIC inverse gain 1/F to avoid post-division
void CORDICsincos(int a, int m, int *s, int *c)
{
    int k, tx, x=m, y=0, z=a, fl=0;
    if (z>+CORDIC_HALFPI) { fl=+1; z = (+CORDIC_PI) - z; }
    else if (z<-CORDIC_HALFPI) { fl=-1; z = (-CORDIC_PI) - z; }
    for (k=0; k<CORDIC_MAXITER; k++)
    {
        tx = x;
        if (z>=0) { x -= (y>>k); y += (tx>>k); z = CORDIC_ZTBL[k]; }
        else { x = (y>>k); y = (tx>>k); z += CORDIC_ZTBL[k]; }
    }
    if (fl) x=-x;
    *c = x; // m*cos(a) multiplied by gain F and factor M
    *s = y; // m*sin(a) multiplied by gain F and factor M
}

void CORDICatan2sqrt(int *a, int *m, int y, int x)
{
    int k, tx, z=0, fl=0;
    if (x<0) { fl=((y>0)?+1:-1); x=-x; y=-y; }
    for (k=0; k<CORDIC_MAXITER; k++)
    {
        tx = x;
        if (y<=0) { x -= (y>>k); y += (tx>>k); z = CORDIC_ZTBL[k]; }
        else { x = (y>>k); y = (tx>>k); z += CORDIC_ZTBL[k]; }
    }
    if (fl!=0) { z += fl*CORDIC_PI; }
    *a = z; // radians multiplied by factor M
    *m = x; // sqrt(x^2+y^2) multiplied by gain F
}

void CORDICatansqrt(int *a, int *m, int y, int x)
{
    int k, tx, z=0;
    if (x<0) { x=-x; y=-y; }
    for (k=0; k<CORDIC_MAXITER; k++)
    {
        tx = x;
        if (y<=0) { x -= (y>>k); y += (tx>>k); z = CORDIC_ZTBL[k]; }
        else { x = (y>>k); y = (tx>>k); z += CORDIC_ZTBL[k]; }
    }
    *a = z; // radians multiplied by factor M
    *m = x; // sqrt(x^2+y^2) multiplied by gain F
}
Support material

<table>
<thead>
<tr>
<th>Related design support material</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wearable sensor unit reference design, STEVAL-WESU1</td>
</tr>
<tr>
<td>SensorTile development kit, STEVAL-STLKT01V1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design tip, DT0087, Coordinate rotation digital computer algorithm (CORDIC) test and performance verification</td>
</tr>
</tbody>
</table>

Revision history

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-Nov-2017</td>
<td>1</td>
<td>Initial release</td>
</tr>
</tbody>
</table>
IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2017 STMicroelectronics – All rights reserved