DT0087
Design tip

Coordinate rotation digital computer algorithm (CORDIC) test and performance verification

By Andrea Vitali

Main components	
STM32L031C4/E4/F4/G4/K4 STM32L031C6/E6/F6/G6/K6	Access line ultra-low-power 32-bit MCU Arm®-based Cortex®-M0+, up to 32 Kbytes Flash, 8 Kbytes SRAM, 1 Kbyte EEPROM, ADC
STM32F031C4/F4/G4/K4 STM32F031C6/E6/F6/G6/K6	Arm®-based 32-bit MCU with up to 32 Kbytes Flash, 9 timers, ADC and communication interfaces, 2.0 - 3.6 V

Purpose and benefits

This design tip explains how to test and evaluate the performance of the coordinate rotation digital computer algorithm (CORDIC).

- Reference look-up-tables (LUTs) are given for all coordinate systems: circular, linear and hyperbolic.
- A utility is provided to test all CORDIC modes, rotation mode and vectoring mode, for all coordinate systems.
- A utility is provided to test the CORDIC implementation for the circular coordinate system specialized for trigonometric functions: sin() and cos(), atan() and sqrt(), atan2().

The implementation is given in the companion Design tip, DT0085, “Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions”.

Description

For CORDIC testing, LUTs have been generated by the tool provided in the companion Design tip to allow 30-bit accuracy (9 digits, 1e-8 min error). The scaling factor has been chosen to ease the CORDIC implementation as pi/2, pi and 2pi are all powers of two.

CORDICtable.c LUT for circular coordinate system

```
//CORDIC, 30 bits, 28 iterations
// 1.0 = 85445659.447054 multiplication factor
// A   = 1.743287 convergence angle (limit is 1.7432866 = 99.9deg)
// F   = 1.646760 gain (limit is 1.64676025812107)
// 1/F = 0.607253 inverse gain (limit is 0.607252935008881)
// pi  = 3.141593 (3.1415926536897932384626)

#define CORDIC_A        1.743287 // CORDIC convergence angle A
#define CORDIC_F        0x08630AA4 // CORDIC gain F
#define CORDIC_1F       0x0317BC17 // CORDIC inverse gain 1/F
#define CORDIC_HALFPI   0x08000000
#define CORDIC_PI       0x10000000
```

```
#define CORDIC_TWOPI    0x20000000
#define CORDIC_MUL      85445659.447054 // CORDIC multiplication factor M = 2^28/pi
#define CORDIC_MAXITER  28

int CORDIC_ZTBL[] = {
  0x04000000, 0x025C80A4, 0x013F670B, 0x00A2223B, 0x005161A8, 0x0028BAFC, 0x00145EC4, 0x000A2F8B,
  0x000517CA, 0x00028BE6, 0x000145F3, 0x0000A2FA, 0x0000517D, 0x000028BE, 0x0000145F, 0x00000A30,
  0x00000518, 0x0000028C, 0x00000146, 0x000000A3, 0x00000051, 0x00000029, 0x00000014, 0x0000000A,
  0x00000005, 0x00000003, 0x00000001, 0x00000001 };
```

CORDIC.c for circular coordinate system

```
#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
}
```

CORDICtable.c LUT for linear coordinate system

```
//CORDIC_LIN, 30 bits, 28 iterations
// 1.0 = 85445659.447054 multiplication factor
// A   = 2.000000 convergence angle (limit is 2)
// F   = 1.000000 gain (limit is 1.0)
// 1/F = 1.000000 inverse gain (limit is 1.0)
// pi  = 3.141593 (3.1415926536897932384626)

#define CORDIC_LIN_F        0x0517CC1B // CORDIC gain F
#define CORDIC_LIN_1F       0x0517CC1B // CORDIC inverse gain 1/F
#define CORDIC_LIN_HALFPI   0x08000000
#define CORDIC_LIN_PI       0x10000000
#define CORDIC_LIN_TWOPI    0x20000000
#define CORDIC_LIN_MUL      85445659.447054 // CORDIC multiplication factor M = 2^28/pi
#define CORDIC_LIN_MAXITER  28

int CORDIC_LIN_ZTBL[] = {
  0x0517CC1B, 0x028BE60E, 0x0145F307, 0x00A2F983, 0x00517CC2, 0x0028BE61, 0x00145F30, 0x000A2F98,
  0x000517CC, 0x00028BE6, 0x000145F3, 0x0000A2FA, 0x0000517D, 0x000028BE, 0x0000145F, 0x00000A30,
  0x00000518, 0x0000028C, 0x00000146, 0x000000A3, 0x00000051, 0x00000029, 0x00000014, 0x0000000A,
  0x00000005, 0x00000003, 0x00000001, 0x00000001 };
```

CORDIC.c for linear coordinate system

```
#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
}
```

CORDICtable.c LUT for hyperbolic coordinate system

```
//CORDIC_HYPER, 30 bits, 28 iterations
// 1.0 = 85445659.447054 multiplication factor
// A   = 1.118173 convergence angle (limit is 1.1181730 = 64.0deg)
// F   = 0.828159 gain (limit is 0.82978162013890)
// 1/F = 1.207497 inverse gain (limit is 1.20513635844646)
// pi  = 3.141593 (3.14159265368979323846 26)

#define CORDIC_HYPER_F        0x0437C07F // CORDIC gain F
#define CORDIC_HYPER_1F       0x062654D7 // CORDIC inverse gain 1/F
#define CORDIC_HYPER_HALFPI   0x08000000
#define CORDIC_HYPER_PI       0x10000000
#define CORDIC_HYPER_TWOPI    0x20000000
#define CORDIC_HYPER_MUL      85445659.447054 // CORDIC multiplication factor M = 2^28/pi
#define CORDIC_HYPER_MAXITER  28

int CORDIC_HYPER_ZTBL[] = {
  0x80000000, 0x02CC2F12, 0x014D01AC, 0x00A3D4E0, 0x005197FC, 0x0028C1C7, 0x00145F9D, 0x000A2FA6,
  0x000517CE, 0x00028BE6, 0x000145F3, 0x0000A2FA, 0x0000517D, 0x000028BE, 0x0000145F, 0x00000A30,
  0x00000518, 0x0000028C, 0x00000146, 0x000000A3, 0x00000051, 0x00000029, 0x00000014, 0x0000000A,
  0x00000005, 0x00000003, 0x00000001, 0x00000001 };
```

CORDIC.c for hyperbolic coordinate system

```
#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 test utility

The output includes the true value of each function and the error of the CORDIC output with respect to its true value. With the CORDIC tables provided above, the error is 5e-8 on average.

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

#include "CORDIC.c"
#include "CORDIC_LIN.c"
#include "CORDIC_HYPER.c"

#define RADSTEP 0.2

int main(int argc, char *argv[]) {
  int x,y,z,xx,yy,zz;
  double a,xxd,yyd,zzd;

  //---- CORDIC circular
  printf("\n---- CORDIC rotation (sin and cos)\n");
  for(a=-1.74;a<+1.74;a+=RADSTEP) {
    x=CORDIC_1F; y=0; z=(int)round(a*CORDIC_MUL);
    CORDIC_rotation_Zto0(x,y,z,&xx,&yy);
    xxd=(double)xx/CORDIC_MUL; yyd=(double)yy/CORDIC_MUL;
    printf("%+f cos:%+f %+e sin:%+f %+e\n",a,xxd,xxd-cos(a),yyd,yyd-sin(a));
  }
  printf("\n---- CORDIC vectoring (sqrt and atan2)\n");
  for(a=-1.74;a<+1.74;a+=RADSTEP) {
    x=(int)round(cos(a)*CORDIC_MUL); y=(int)round(sin(a)*CORDIC_MUL); z=0;
    CORDIC_vectoring_Yto0(x,y,z,&xx,&zz);
    xxd=(double)xx/CORDIC_F; zzd=(double)zz/CORDIC_MUL;
```

```
    printf("%+f sqrt:%+f %+e atan:%+f %+e\n",a,xxd,xxd-1.0,zzd,zzd-a);
  }

  //---- CORDIC linear
  printf("\n---- CORDIC LIN rotation (mul)\n");
  for(a=-2;a<+2;a+=RADSTEP) {
    x=(int)round(2.0*a*CORDIC_LIN_MUL); y=0; z=(int)round(a*CORDIC_LIN_MUL);
    CORDIC_LIN_rotation_Zto0(x,y,z,&xx,&yy);
    yyd=(double)yy/CORDIC_LIN_MUL;
    printf("%+f mul:%+f %+e\n",a,yyd,yyd-2.0*a*a);
  }
  printf("\n---- CORDIC LIN vectoring (div)\n");
  for(a=-4;a<+4;a+=RADSTEP) {
    x=(int)round((2.5)*CORDIC_LIN_MUL); y=(int)round((a)*CORDIC_LIN_MUL); z=0;
    CORDIC_LIN_vectoring_Yto0(x,y,z,&xx,&zz);
    zzd=(double)zz/CORDIC_LIN_MUL;
    printf("%+f div:%+f %+e\n",a,zzd,zzd-a/2.5);
  }

  //---- CORDIC hyperbolic
  printf("\n---- CORDIC HYPER rotation (sinh and cosh)\n");
  for(a=-1.11;a<+1.11;a+=RADSTEP) {
    x=CORDIC_HYPER_1F; y=0; z=(int)round(a*CORDIC_HYPER_MUL);
    CORDIC_HYPER_rotation_Zto0(x,y,z,&xx,&yy);
    xxd=(double)xx/CORDIC_HYPER_MUL; yyd=(double)yy/CORDIC_HYPER_MUL;
    printf("%+f cosh:%+f %+e sinh:%+f %+e\n",a,xxd,xxd-cosh(a),yyd,yyd-sinh(a));
  }
  printf("\n---- CORDIC HYPER rotation (exp)\n");
  for(a=-1.11;a<+1.11;a+=RADSTEP) {
    x=CORDIC_HYPER_1F; y=0; z=(int)round(a*CORDIC_HYPER_MUL);
    CORDIC_HYPER_rotation_Zto0(x,y,z,&xx,&yy);
    xxd=(double)xx/CORDIC_HYPER_MUL; yyd=(double)yy/CORDIC_HYPER_MUL;
    printf("%+f exp:%+f %+e\n",a,xxd+yyd,(xxd+yyd)-exp(a));
  }
  printf("\n---- CORDIC HYPER vectoring (sqrt and atanh)\n");
  for(a=-1.11;a<+1.11;a+=RADSTEP) {
    x=(int)round(cosh(a)*CORDIC_HYPER_MUL); y=(int)round(sinh(a)*CORDIC_HYPER_MUL); z=0;
    CORDIC_HYPER_vectoring_Yto0(x,y,z,&xx,&zz);
    xxd=(double)xx/CORDIC_HYPER_F; zzd=(double)zz/CORDIC_HYPER_MUL;
    printf("%+f sqrt:%+f %+e atanh:%+f %+e\n",a,xxd,xxd-1.0,zzd,zzd-a);
  }
  printf("\n---- CORDIC HYPER vectoring (sqrt and ln)\n");
  for(a=RADSTEP;a<+8;a+=RADSTEP) {
    x=(int)round((a+1.0)*CORDIC_HYPER_MUL); y=(int)round((a-1.0)*CORDIC_HYPER_MUL); z=0;
    CORDIC_HYPER_vectoring_Yto0(x,y,z,&xx,&zz);
    xxd=(double)xx/CORDIC_HYPER_F/2.0; zzd=(double)zz/CORDIC_HYPER_MUL*2.0;
    printf("%+f sqrt:%+f %+e ln:%+f %+e\n",a,xxd,xxd-sqrt(a),zzd,zzd-log(a));
  }

  return 0;
}
```

CORDIC test for circular coordinates, specialized for trigonometric functions

CORDICtable.c LUT for circular coordinate system, specialized for trigonometric functions

LUT has been generated by the tool provided in the companion Design tip to allow 16-bit accuracy (4-5 digits, 1e-3 to 1e-4 error). The scaling factor has been chosen to ease the final scaling as the CORDIC multiplication factor is a power of two.

```
//CORDIC, 16 bits, 14 iterations
// 1.0 = 8192.000000 multiplication factor
// A   = 1.743165 convergence angle (limit is 1.7432866 = 99.9deg)
// F   = 1.646760 gain (limit is 1.64676025812107)
// 1/F = 0.607253 inverse gain (limit is 0.607252935008881)
// pi  = 3.141593 (3.1415926536897932384626)

#define CORDIC_A        1.743165 // CORDIC convergence angle A
#define CORDIC_F        0x000034B2 // CORDIC gain F
#define CORDIC_1F       0x0000136F // CORDIC inverse gain 1/F
#define CORDIC_HALFPI   0x00003244
#define CORDIC_PI       0x00006488
#define CORDIC_TWOPI    0x0000C910
#define CORDIC_MUL      8192.000000 // CORDIC multiplication factor M = 2^13
#define CORDIC_MAXITER  14

int CORDIC_ZTBL[] = {
  0x00001922, 0x00000ED6, 0x000007D7, 0x000003FB, 0x000001FF, 0x00000100, 0x00000080,
0x00000040,
  0x00000020, 0x00000010, 0x00000008, 0x00000004, 0x00000002, 0x00000001 };
```

CORDIC.c for circular coordinate system, specialized for trigonometric functions

```
#include "CORDICtable_specialized.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
}
```

CORDIC test utility, specialized for trigonometric functions

This utility is run by the MATLAB® script included in the next paragraph.

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

#include "CORDIC_specialized.c"

int main(int argc, char *argv[]) {
  double aref, mref, sref, cref;
  double adbl, mdbl, sdbl, cdbl;
  int    a,    m,    s,    c,    i;

  // test sin cos
  for(i=-180;i<=+180;i++) {
    aref = (double)i*M_PI/180; mref = 1.0;
    cref = mref * cos(aref); sref = mref * sin(aref);
    a = (int)round(aref*CORDIC_MUL);
    m = CORDIC_1F; // to avoid post-div/mul by CORDIC_K/1K
    CORDICsincos(a,m,&s,&c);
   sdbl = ((double)s/CORDIC_MUL); cdbl = ((double)c/CORDIC_MUL);
    printf("%d %f %f %e %f %f %e\n",i,sref,sdbl,sref-sdbl,cref,cdbl,cref-cdbl);
  }

  // test atan2 sqrt
  for(i=-180;i<=+180;i++) {
    aref = (double)i*M_PI/180; mref = 1.0; sref=sin(aref); cref=cos(aref);
    s = (int)round(sref*CORDIC_MUL); c = (int)round(cref*CORDIC_MUL);
    CORDICatan2sqrt(&a,&m,s,c);
    adbl = ((double)a/CORDIC_MUL); if(i==180) adbl+=2.0*M_PI; // adbl is near -pi, should be
near +pi
    mdbl = ((double)m/CORDIC_F);
    printf("%d %f %f %e %f %f %e\n",i,aref,adbl,aref-adbl,mref,mdbl,mref-mdbl);
  }

  // test atan sqrt
  for(i=-180;i<=+180;i++) {
    aref = (double)i*M_PI/180; mref = 1.0;
    sref=sin(aref); cref=cos(aref);
    if(i<-90) aref=(double)(i+180)*M_PI/180;
    if(i>+90) aref=(double)(i-180)*M_PI/180;
    s = (int)round(sref*CORDIC_MUL); c = (int)round(cref*CORDIC_MUL);
    CORDICatansqrt(&a,&m,s,c);
    adbl = ((double)a/CORDIC_MUL); mdbl = ((double)m/CORDIC_F);
    printf("%d %f %f %f %f %f %f\n",i,aref,adbl,aref-adbl,mref,mdbl,mref-mdbl);
  }
```

```
  return 0;
}
```

CORDIC MATLAB® test script, specialized for trigonometric functions

This MATLAB® script runs the C utility included in the previous paragraph. The corresponding plots are shown in Figures 1-3.

```
system('CORDIC_specialized_test >CORDIC_specialized_test_out.txt');
data=importdata('CORDIC_specialized_test_out.txt');
sz=size(data); L=sz(1)/3;

%--- test sincos
t1angle_deg=data(1:L,1); t=t1angle_deg;
t1sinref   =data(1:L,2); t1sinCORDIC=data(1:L,3); t1sinerr=data(1:L,4);
t1cosref   =data(1:L,5); t1cosCORDIC=data(1:L,6); t1coserr=data(1:L,7);

figure;
subplot(2,1,1); hold on;
plot(t,t1sinref,'o'); plot(t,t1cosref,'o');
plot(t,t1sinCORDIC,'x'); plot(t,t1cosCORDIC,'x');
legend('sin ref','cos ref','sin CORDIC','cos CORDIC');
grid on; zoom on; title('sincos'); xlabel('deg');
subplot(2,1,2); hold on;
plot(t,t1sinerr,'^'); plot(t,t1coserr,'v');
legend('sin err','cos err'); grid on; zoom on; xlabel('deg');
title(sprintf('sincos error, max %g',max(abs([t1sinerr;t1coserr]))));

%--- test atan2 sqrt
t2angle_deg   =data(L+1:2*L,1); t=t2angle_deg;
t2angleref    =data(L+1:2*L,2); t2angleCORDIC=data(L+1:2*L,3); t2angleerr=data(L+1:2*L,4);
t2modref      =data(L+1:2*L,5); t2modCORDIC   =data(L+1:2*L,6); t2moderr   =data(L+1:2*L,7);

figure;
subplot(2,2,1); hold on; plot(t,t2angleref,'o'); plot(t,t2angleCORDIC,'x');
legend('angle ref','angle CORDIC');
grid on; zoom on; title('atan2(y,x)'); xlabel('deg');
subplot(2,2,2); hold on; plot(t,t2modref,'o'); plot(t,t2modCORDIC,'x');
legend('mod ref','mod CORDIC');
grid on; zoom on; title('sqrt'); xlabel('deg');
subplot(2,2,3); plot(t,t2angleerr,'.');
legend('angle err'); xlabel('deg'); grid on; zoom on;
title(sprintf('atan2 err, max %g',max(abs(t2angleerr))));
subplot(2,2,4); plot(t,t2moderr,'.');
legend('mod err'); xlabel('deg'); grid on; zoom on;
title(sprintf('sqrt err, max %g',max(abs(t2moderr))));

%--- test atan sqrt
t3angle_deg   =data(2*L+1:3*L);   t=t3angle_deg;
t3angleref    =data(2*L+1:3*L,2); t3angleCORDIC=data(2*L+1:3*L,3); t3angleerr=data(2*L+1:3*L,4);
t3modref      =data(2*L+1:3*L,5); t3modCORDIC   =data(2*L+1:3*L,6); t3moderr   =data(2*L+1:3*L,7);

figure;
subplot(2,2,1); hold on; plot(t,t3angleref,'o'); plot(t,t3angleCORDIC,'x');
legend('angle ref','angle CORDIC');
grid on; zoom on; title('atan(y/x)'); xlabel('deg');
subplot(2,2,2); hold on; plot(t,t3modref,'o'); plot(t,t3modCORDIC,'x');
legend('mod ref','mod CORDIC');
grid on; zoom on; title('sqrt'); xlabel('deg');
subplot(2,2,3); plot(t,t3angleerr,'.');
legend('angle err'); xlabel('deg'); grid on; zoom on;
title(sprintf('atan err, max %g',max(abs(t3angleerr))));
subplot(2,2,4); plot(t,t3moderr,'.');
legend('mod err'); xlabel('deg'); grid on; zoom on;
title(sprintf('sqrt err, max %g',max(abs(t3moderr))));
```

Figure 1. Test of Sin() and Cos() in the specialized CORDIC

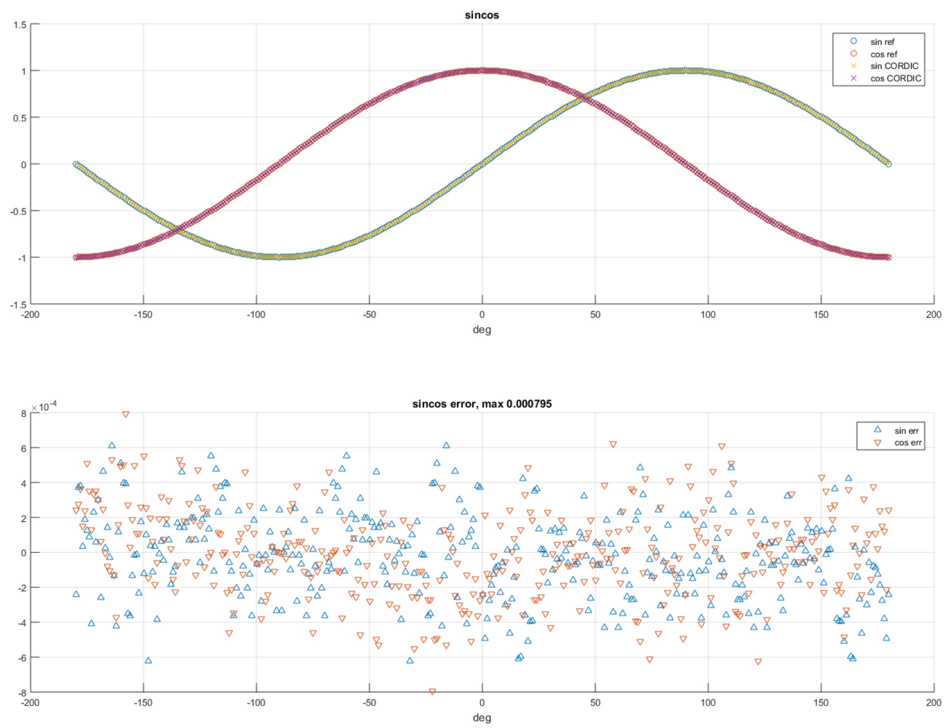


Figure 2. Test of Atan() and Sqrt() function in the specialized CORDIC

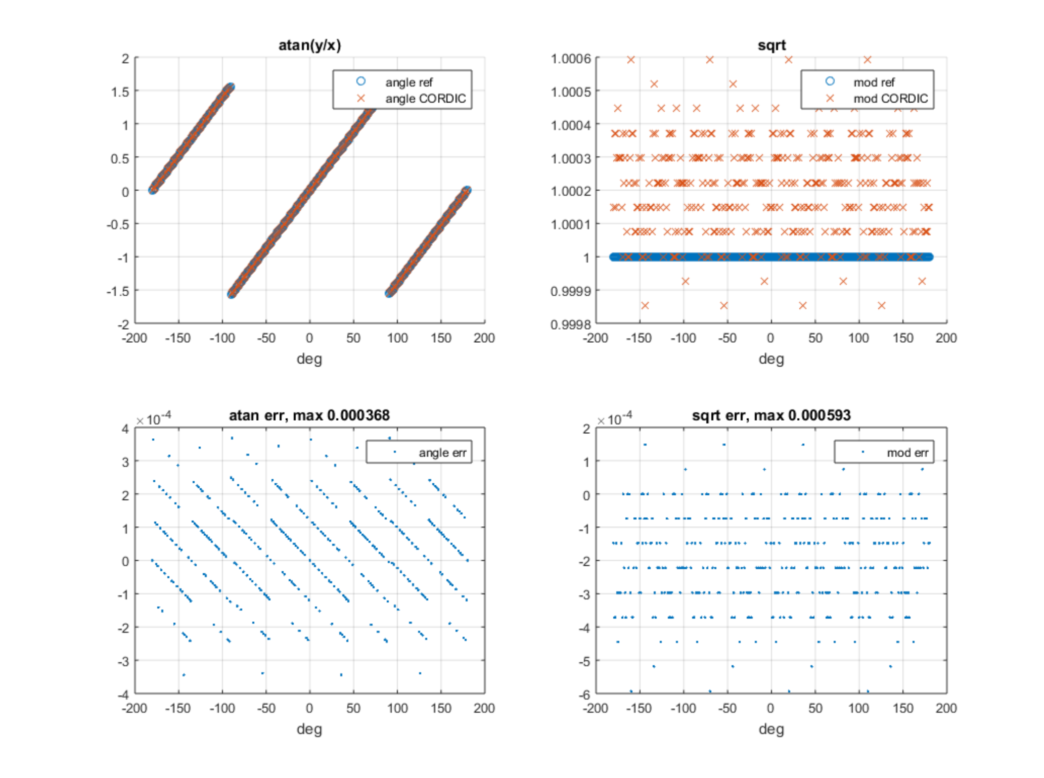
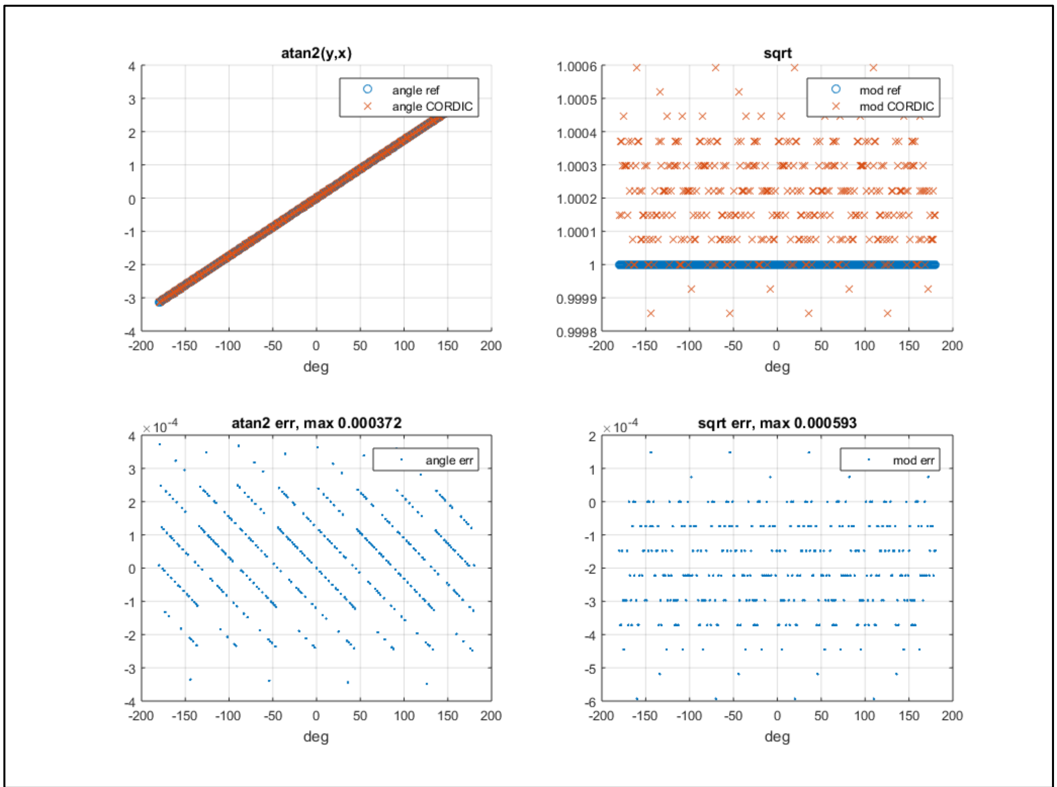


Figure 3. Test of Atan2() and Sqrt() function in the specialized CORDIC



CORDIC benchmarks on Cortex-M4F and Cortex-M0

The benchmark is about computing the sin() and cos() functions on a Cortex-M4F (with FPU active) and on a Cortex-M0 (without FPU). The fewer the clock cycles, the better. The longest time was obtained with no code optimization. The shortest time was obtained with compiler optimization for speed (gcc -o3).

Sin() and Cos()	Cortex-MF4	Cortex-M0	Notes
IEEE double float	2720 (10T)	10400 (30T)	2e-16 min error
IEEE single float	360 (T ref.)	7700 (20T)	1e-7 min error
CORDIC int32	900-300 (3T-T)	1200-280 (4T-T)	3e-6 max error
Polynomial int32	370-140 (T-T/2)	430-190 (T-T/2)	6e-6 max error
ARM CMSIS	370-110 (T-T/3)	n/a	+220kB code size

The CORDIC table was generated to allow 30-bit accuracy (9 digits, 1e-8 min error) but iterations were limited from 28 to 19 in order to match the error of the polynomial approximation.

The polynomial approximation for sin() and cos() is based on the sum or difference of a 6^{th} and 5^{th} order polynomial with int32 coefficients.

The ARM CMSIS library could not be compiled on the Cortex-M0 because the available Flash memory was insufficient.

Support material

Related design support material
Wearable sensor unit reference design, STEVAL-WESU1
SensorTile development kit, STEVAL-STLKT01V1
Documentation
Design tip, DT0085, Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions

Revision history

Date	Version	Changes
16-Nov-2017	1	Initial release.