Simple C source for CORDIC

CORDIC is a simple and effecient algorithm computing the sine and cosine of a value using only basic arithmetic (addition, subtraction and shifts). Below is some very simple ANSI C code for fixed point CORDIC calculations. It is based on the definitions given in the excellent FXTBook . Read that if you're interested in more detail.

Code

The program generates a header file with th constants and code for CORDIC computations when using different word sizes. It will write the header file to standard output. Note that this uses the math library to compute the constants -- so run it on a device with an FPU, then compile the resulting H file on your target device if you are cross-compiling onto another platform (e.g. onto a microcontroller). Specify the number of bits to be used -- this will generate tables which map 1.0 == 2^(bitsize-2). For 16-bits for example, the resulting header file will use a 2.14 fixed-point representation. This will ensure maximum precision. You could alter **mul** below to be another value if you already have a fixed-point representation you wish to use.

gentable.c

```
#include <math.h>
#define M_PI 3.14159265358979323846264
#define K1 0.607252935008881256169 4

int main(int argc, char **argv)
{
    int i;
    int bits = 32; // number of bits
    int mul = (1<<(bits-2));


    int n = bits; // number of elements.
    int c;

    printf("//Cordic in %d bit signed fixed point math\n", bits);
    printf("//Function is valid for arguments in range -pi/2 -- pi/2\n");
    printf("//for values pi/2--pi: value = half_pi-(theta-half_pi) and similarly for values -pi---pi/2\n");
    printf("//\n");
    printf("// 1.0 = %d\n", mul);
    printf("// 1/k = 0.607252935008881256169 4\n");
    printf("// pi = 3.14159265358979323846264\n");

    printf("//Constants\n");
    printf("#define cordic_1K 0x%08X\n", (int)(mul*K1));
    printf("#define half_pi 0x%08X\n", (int)(mul*(M_PI/2)));
    printf("#define MUL %f\n", (double)mul);
    printf("#define CORDIC_NTAB %d\n", n);

    printf("int cordic_ctab [] = {");
    for(i=0;i<n;i++)
    {
        c = (atan(pow(2, -i)) * mul);
        printf("0x%08X, ", c);
    }
    printf("};\n\n");

    //Print the cordic function
    printf("void cordic(int theta, int *s, int *c, int n)\n{\n  int k, d, tx, ty, tz;\n");
    printf("  int x=cordic_1K,y=0,z=theta;\n  n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;\n");
    printf("  for (k=0; k<n; ++k)\n  {\n    d = z>>%d;\n", (bits-1));
    printf("    //get sign. for other architectures, you might want to use the more portable version\n");
    printf("    //d = z>=0 ? 0 : -1;\n    tx = x - (((y>>k) ^ d) - d);\n    ty = y + (((x>>k) ^ d) - d);\n");
    printf("    tz = z - ((cordic_ctab[k] ^ d) - d);\n    x = tx; y = ty; z = tz;\n  }  \n *c = x; *s = y;\n}\n");
```

```
}
```

The CORDIC function will expect a value from -pi/2 to pi/2. Other values can be computed easily. For pi/2--pi, the result is just (pi/2-(a-pi/2)) and similarly for -pi---pi/2. Every other angle can be mapped into this range by taking the angle mod pi. Assuming you set bits to 32, and saved the output from gentable.c to "cordic-32bit.h", you will have a file like this:

cordic-32bit.h

```
//Cordic in 32 bit signed fixed point math
//Function is valid for arguments in range -pi/2 -- pi/2
//for values pi/2--pi: value = half_pi-(theta-half_pi) and similarly for values -pi---pi/2
//
// 1.0 = 1073741824
// 1/k = 0.6072529350088812561694
// pi = 3.1415926535897932384626
//Constants
#define cordic_1K 0x26DD3B6A
#define half_pi 0x6487ED51
#define MUL 1073741824.000000
#define CORDIC_NTAB 32
int cordic_ctab [] = {0x3243F6A8, 0x1DAC6705, 0x0FADBAFC, 0x07F56EA6, 0x03FEAB76, 0x01FFD55B,
0x00FFFAAA, 0x007FFF55, 0x003FFFEA, 0x001FFFFD, 0x000FFFFF, 0x0007FFFF, 0x0003FFFF,
0x0001FFFF, 0x0000FFFF, 0x00007FFF, 0x00003FFF, 0x00001FFF, 0x00000FFF, 0x000007FF,
0x000003FF, 0x000001FF, 0x000000FF, 0x0000007F, 0x0000003F, 0x0000001F, 0x0000000F,
0x00000008, 0x00000004, 0x00000002, 0x00000001, 0x00000000, };

void cordic(int theta, int *s, int *c, int n)
{
  int k, d, tx, ty, tz;
  int x=cordic_1K,y=0,z=theta;
  n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;
  for (k=0; k<n; ++k)
  {
    d = z>>31;
    //get sign. for other architectures, you might want to use the more portable version
    //d = z>=0 ? 0 : -1;
    tx = x - (((y>>k) ^ d) - d);
    ty = y + (((x>>k) ^ d) - d);
    tz = z - ((cordic_ctab[k] ^ d) - d);
    x = tx; y = ty; z = tz;
  }
 *c = x; *s = y;
}
```

Obviously a 16 bit CORDIC algorithm can be created just as easily (the result is cordic-16bit.h in this case). The following file shows the CORDIC computation with a simple test function that compares the results to the standard math implementation.

cordic-test.c

```
#include "cordic-32bit.h"
#include <math.h> // for testing only!

//Print out sin(x) vs fp CORDIC sin(x)
int main(int argc, char **argv)
{
    double p;
    int s,c;
    int i;
    for(i=0;i<50;i++)
    {
        p = (i/50.0)*M_PI/2;
        //use 32 iterations
        cordic((p*MUL), &s, &c, 32);
        //these values should be nearly equal
        printf("%f : %f\n", s/MUL, sin(p));
```

```
        }
    }
```

The CORDIC algorithm is not perfectly accurate, but it simple to implement on limited hardware devices (those without hardware multipliers or with very little storage space).