

BASIC Stamp II math note #C

BASIC Stamp, CORDIC math implementation

(c) 2002,2005 EME Systems, Berkeley CA U.S.A.

Tracy Allen

[<stamp index>](#) [<home>](#)

contents (updated 12/28/2005)

Demo programs for CORDIC math:

[CORDIC-atn.bpe](#)

Given x and y, computes the angle and the length of the vector from the origin to point (x,y). (revised 12/05)

[CORDIC-sincos.bpe](#)

Given an angle in degrees, 0-360, calculates the sine and cosine to x.xxx.

CORDIC is an acronym for **COordinate Rotation Digital Computer**. The notions behind this computing machinery were motivated by the need to calculate the trig functions and inverse trig functions in real time navigation systems. The CORDIC algorithms require only shifts, adds and table lookups, simple integer math. So, it is possible to implement a specialized CORDIC machine, small and fast enough for real time calculations, dedicated to that one purpose. The algorithm can be coded in firmware on even small microcontrollers. With slight modifications in initial conditions and data tables, the core algorithm can multiply, divide, modulate, and calculate square roots, hyperbolic functions, exponentials and logs. It is this versatility and simplicity, that make CORDIC the preferred implementation of math functions on small hand calculators.

The purpose of this note is to show how to code the algorithms BASIC Stamp. First off, it is an educational exercise, to see how these fascinating algorithms work. Beyond that, sometimes you really need one of these math functions. The alternative is often a table lookup with interpolation, an external coprocessor, a different algorithm (which may turn out to be quite beyond the capability of the Stamp), or a rethinking of what is really required. The BASIC Stamp of course has no floating point unit, although you can add one as an external chip. The same can be said for most small embedded processors, like the PIC or the SX series, or the MSP430 series. The CORDIC method is attractive because it is an integer algorithm close to the hardware. Of course the Stamp will not be fast in its implementation, but in some applications it is fast enough.

You will find very little about CORDIC in math books or other literature. For example, Jack Crenshaw does not mention it in his excellent book, [Math Toolkit for Real Time Programming](#), nor is there a single mention of it in Donald Knuth's seminal tome, [The Art of Computer Programming](#). I don't know why. Maybe the algorithm is too unsophisticated or so specialized as to be relegated to hand calculators, coprocessors and navigational engines. The computational methods of choice, explained in detail in college math courses and texts, are based on power series expansions of the functions, or solvers such as Newton's method. The series expansion for each mathematical function can present unique difficulties in terms of convergence, so each function ends up with a different piece of computer code, different from the next function in the library. They run best on a full general purpose computer with floating point math capabilities built into the hardware, and

under such conditions they can be optimized computational speed and accuracy. But for generality and for compatibility with simple integer hardware, the CORDIC method has no peer.

Here are some links to URLs relating to the CORDIC method (also try Google, I know some of these are out of date):

[Ray Andraka, Consulting](#)

gurus' guru. By far the most incisive discussion of the subject I know of. Download the pdf file, "Survey of CORDIC Algorithms for FPGAs".

[Grant R. Griffin](#)

dspGURU. An informative FAQ, with additional links to the online and printed literature.

[Ingo Cyliax](#)

CORDIC swiss army knife for computing math functions.. An article from Circuit Cellar INK. It even includes a program for calculating sine and cosine on the [BASIC Stamp II](#). A different approach from the one taken here.

[Pitts Jarvis](#)

"IMPLEMENTING CORDIC ALGORITHMS A single compact routine for computing transcendental functions" An tutorial from Dr. Dobbs Journal, October 1990. [code](#)

[Helmut Knaust](#)

"How Do Calculators Calculate? A tutorial from University of Texas.

Volder, J.E., 1959, The CORDIC Trigonometric Computing Technique, IRE Transactions on Electronic Computers, V. EC-8, No. 3, pp. 330-334

seminal paper

[Vladimir Baykov](#)

Interesting newsgroup posting by Vladimir BAYKOV

[Norbert Lindlbauer, Berkeley Center for New Music](#)

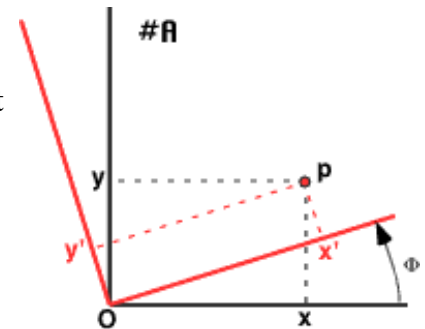
Discussion of CORDIC architectures with a view toward music synthesizer

Givens transform, the underpinning of CORDIC

[top](#)

The CORDIC method depends on the Givens rotation transform, which expresses the coordinates of a point P in two different coordinate systems, rotated by a certain angle about their common origin. In the first system point P has coordinates (x,y) and in the second system the same point P has coordinates (x',y'). The second system is rotated by an angle H with relation to the first. (H=phi in the diagram to the right-sorry about the notation). The relation between the coordinates is:

$$\begin{aligned}x' &= x \cos H - y \sin H \\y' &= y \cos H + x \sin H\end{aligned}$$



Look at a [derivation of the Givens transform](#) which illustrates the above formula with a geometrical proof. It also shows the close relation between coordinate rotation and vector rotation.

CORDIC is a successive approximation algorithm. A sequence of successively smaller rotations based on binary decisions hone in on the value we want to find. For example, we start with initial values of x and y, and end up with what we want, the angle whose tangent is y/x and also with the length of the vector $\text{SQR}(x^2+y^2)$.

Engineers are familiar with successive approximation, where, say, an unknown voltage is presented at one input, and is first compared with 1/2 of a reference voltage. If the unknown is greater than 1/2 the reference,

then the most significant bit of the answer is 1 and the next value to compare with is 3/4 of the reference, otherwise, the most significant bit of the answer is 0 and the next value to compare with is 1/4 of the reference. The sequence of decisions up=1 and down=0 at each step generates the digital binary output for as many steps as the system precision will allow.

In CORDIC, the comparison takes place between the one of the variables and a target value for that variable.

For example, suppose you start with a given angle, and what you want to find is the sine and cosine of that angle. Assume there is an initial unit vector on the x axis. The first decision is whether the given angle is greater than or less than zero. The first rotation is accordingly then either ($\tan H = +1$) or ($\tan H = -1$), that is 45 degrees positive or negative. The angle accumulator now contains either +45 degrees or -45 degrees. The original vector is rotated to that angle with new coordinates ($x'y'$). The next rotation is ($\tan H = \pm 1/2$) in the direction that rotates toward the given angle. All the computer has to decide is the direction, based on comparing the value in the angle accumulator with the starting angle. The angle accumulator at each step is updated by lookup of the corresponding $\arctan(H)$ in the table, added to or subtracted from the current value in the angle accumulator. Angles add up. The net rotation is the simple sum of all the small rotations. At each step the coordinates x',y' are updated by means of the Givens transform. Through a sequence of smaller and smaller steps the algorithm ends up with original unit vector rotated so that it coincides with the given starting angle, and the final coordinates x' and y' equal to the sine and cosine of that angle.

Now, suppose you start with a value of y/x , and you want to find the angle that has y/x as its \tan . That is, you want to calculate the arctangent. The coordinates are first rotated by an angle of 45 degrees ($\tan H = \pm 1$) the direction being counterclockwise if $y > 0$ and clockwise if $y < 0$. That determines the most significant bit. Then the next step is a smaller rotation, ($\tan H = \pm 1/2$), also based on the sign of y . The action taken in sequence is to rotate the coordinates in progressively smaller steps until the transformed x axis lines up with the point P , that is, the y coordinate finally is reduced to zero. At each step the computer has to make only the binary decision of clockwise or counter clockwise depending on the current value of the y component. The binary number generated by the sequence of up/down decisions is the unique angle subtended by (x,y) in the original coordinate system. While the y value is reduced to zero in the final coordinate system, the x value is, in the end, and after adjustment by a constant scale factor, equal to the length of the original vector, $\text{SQR}(x^2 + y^2)$.

In the calculation of the arctangent and length of the vector, it is the coordinate y that is minimized, while in the calculation of the sine and cosine, it is the difference between the given angle and the rotated angle that is minimized. Both are CORDIC algorithms, one called "rotation" and the other called "vectoring", and there are many more twists of the same kind for calculation of other kinds of quantities by means of successive rotations. The difference has to do with the starting condition and which variable is minimized. But all involve a successive approximation and a binary decision at each step. The variables are linked through the geometry. Very clever.

Recall that $\tan H = \sin H / \cos H$, so the above formula can be rewritten as,

$$\begin{aligned} x' &= \cos H (x - y \tan H) \\ y' &= \cos H (y + x \tan H) \end{aligned}$$

One neat trick of CORDIC is to allow only angular rotations of $\pm \tan H = 1, 1/2, 1/4, 1/8$ and so on out to the number of bits to be acquired. In that way, the math computation requires only division by powers of two, shifts, and addition or subtraction. The value of each bit in angular base is $\arctan 1/2, \arctan 1/4$ and so on. For example, the translation from the binary value $\theta = 1011$ into standard angular measure is simply $\theta = \arctan(1/2) + -\arctan(1/4) + \arctan(1/8) + \arctan(1/16)$. This computation is usually done at the step by step as the x' and y' are calculated. The angles that correspond to $\text{ATN } 1, \text{ATN } 1/2, \text{ATN } 1/4, \dots$ are stored

in a lookup table in the computer memory, and the angle is accumulated at each step by adding or subtracting the successively smaller values. (Note, the Andraka article points out situations where the funny binary base is itself sufficient, without translation to standard measure.)

Here it is in terms that relate x and y at step i+1, to x and y at step i:

$$\begin{aligned}x(i+1) &= \cos(\text{atn}(1/2^i)) * \{x(i) - y(i) * d(i) * (1/2^i)\} \\y(i+1) &= \cos(\text{atn}(1/2^i)) * \{y(i) + x(i) * d(i) * (1/2^i)\} \\theta &= \theta + d(i) * \arctan(1/2^i)\end{aligned}$$

The iteration starts with i=0. The decision at each step is the factor d(i), which will take on a value of either +1 or - 1, depending on the sign of the decision variable. The particular variable or condition that determines the d(i), that is the main thing that distinguishes one CORDIC algorithm from another. The core calculation is exactly what is written there. Simply by changing the initial conditions and decision criteria, the core algorithm can be made to perform very different tasks. That is what gives it its versatility. Same machinery, many functions.

A second trick, or insight, of CORDIC is that the value of cos H does not depend on the direction of rotation. That is a property of cosine, that $\cos H = \cos -H$. That fact is very convenient, because the cosine terms are a product that does not depend on d(i),

$$\begin{aligned}&\cos(\text{atn}(1)) * \cos(\text{atn}(1/2)) * \cos(\text{atn}(1/4)) * \cos(\text{atn}(1/8)) * \cos(\text{atn}(1/16)) * \dots \\&= 0.707107 * 0.894427 * 0.970143 * 0.992278 * 0.998052 * 0.999512 * 0.999878 * \dots\end{aligned}$$

The value of $\cos(\text{atn}(H))$ approaches unity as H approaches zero, and the limit of the product of terms is a constant number, 0.607252935... It is that many significant digits even after 16 steps. The exact value of the product is easily computed in advance and stored as a constant. This factor is called the CORDIC gain. Some computations (like the length of the vector, or the values of cosine and sine, have to be compensated at some point in the calculation by this gain. That can be done either in the initial conditions or at the end result. In Stamp math, multiplying by 0.607252935 is done as `**39797`, because $39797/65536 \approx 0.607252935$. {Side math note, from simple geometry, $\cos(\text{atn}(1/2^i)) = 1/\sqrt{1 + 2^{-2i}}$ }

scaling for the microcontroller, BASIC Stamp

[top](#)

The BASIC Stamp and microcontrollers in general work with integers. If we want to represent a fraction, we have to assign some integer to be our unity, and then smaller numbers represent fractional parts of that unity.

For example, if number 1 in this system has the value 10000, then the smaller numbers represent fractions from 0/10000, 1/10000, ... , 9999/10000. In some of the other math notes on this site, I have had reason to choose other values for the unity. For example, with a unity of 65536, the fractions are represented in steps of 1/65536, and that is quite a good approximation and is well suited to fractional multiplication using the `**` operator.

To use CORDIC on the BASIC Stamp, a scale has to be chosen for both angles and coordinates.

The choice of units is quite arbitrary, and it can be made to work in any units you want. However, the CORDIC algorithm works only with angles in quadrants 1 and 4 (no matter what units are chosen). Angles outside of that range have to be handled with a wrapper that rotates or reflects the problem into that range and compensates at the end, using standard trigonometric identities.

The BASIC Stamp has built in operators for SIN, COS, ATN and HYP. Those functions represent the circle in units of brads, with 256 brads equal to 360 degrees or 2 pi radians. The angular resolution is thus a little less than one degree. Similarly, SIN and COS take on values from -128 (in twos complement) to +128. Thus those values scale the standard range of SIN and COS from -1 to +1, and the full range is thus capable of a

resolution of about 1/2%.

However, CORDIC can do better with 16 bits to work with. In the programs below, I have chosen to use the number 10000 for the unity for the coordinates, x and y, and the number 16384 to represent the angle 45 degrees. The angles that can be represented on the Stamp thus range from -32768 (-90 degrees) to +32767 (+90 degrees). The example programs show how to provide the wrapper to translate to standard units and how to cover all four quadrants, or a continuous input variable.

The programs depend on a table of data that translates from the CORDIC binary representation of angle to our standard representation, where 45 degrees is 16384. The words in the table are calculated in advance on a calculator or in Excel as

$16384 * \arctan(1)$, $16384 * \arctan(1/2)$, ... , $16384 * \arctan(1/32768)$

In this calculating these value for the table, it will be necessary to round off. This step leads to a loss of precision in the final answer, and for that reason it pays to use as large a number as possible to represent the range of the 1st and 4th quadrant. For example, instead of 16384 to represent 45 degrees, you could use the number 4500 to have the answer come out directly in degrees * 100, or 7854 to have the answer come out in radians * 10000, or 25000 to have the answer come out as a fraction of pi * 100000. But the round off would be more significant when using these smaller numbers. It is better to start with the highest number possible within the 16 bit framework, to optimize the precision, and then rescale at the end. And of course, going to double precision would be best of all, and the principles described here are the same. But the BASIC Stamp is slow and clunky with those calculations, so I'll start simple and leave the double precision for later.

Granted, it is issues of scaling that will take the most consideration in most applications. The CORDIC machinery will seem trivial in comparison, sitting at the core of wrappers upon wrappers.

Specifics of the BASIC Stamp implementation

[top](#)

The demo programs are found at the links at the top of this article, there is one program for calculating the angle and vector length, given a point (x,y), and another program for calculating the sine and cosine of a given angle. The programs make use of some program tricks unique to the BASIC Stamp.

CORDIC makes frequent use of the sign of a number. The decisions on whether to add or subtract are made based on the sign of one particular variable. Negative numbers in the BASIC Stamp are represented in twos complement, where bit 15 is the sign. In these programs each variable defined is accompanied by an alias for its sign bit, as

```
z  VAR Word
zs VAR z.Bit15
```

When it is necessary to calculate a number conditional on its own sign, or on the sign of another number, you will see this kind of sequence.

```
y = y + ( -zs ^ x + zs )
```

The goal of this calculation is to either add or subtract the value of a variable x from another variable y, and the add or subtract depends on the sign of a third variable z. The formula in () follows directly from the definition of twos complement. The negative of a number is its ones complement plus 1. If z is negative, then zs in the above equals 1, and -zs= -1, which in twos complement is all bits equal to 1. The exclusive OR of that with the number x forms the ones complement of x and adding zs=1 completes the twos complement negation. On the other hand, if z is positive, then zs=0, then -zs also equals zero, and the XOR with x and adding zero leaves plain old positive x. Overall, the variable x takes on the sign of z. Often the program uses this in the context of a calculation. For example, division of x by a power of two, 2^I , can be done naively like this.

$y = y + (x \gg i)$ ' divide x by 2^i

Nevertheless that does not work if x is negative, as it often will be during the CORDIC procedure. Instead, this is done this way:

$y = y + (-xs \wedge (ABS\ x \gg i) + xs)$

That is, the division by 2^i , shift right i, is applied to the absolute value, and the sign is restored. It would be possible to do this with IF-THEN statements, but there are so many of these kind of statements that I find that the IF-THEN logic quickly becomes unwieldy. It is also possible to do these divisions by two simply by shifting the value right, and extending the sign. (This works for powers of two, not for division in general) The BASIC Stamp does not provide an easy way to do this, but I think that would be the way to go in machine language implementations. In PBASIC, this comes out as

$y = y + (x \gg i \mid (-xs \ll (16-i)))$

That is, the value of x is shifted right i bits, to divide by 2^i , and then the bits on the left are filled in with the original sign, sign extended.


The attached programs show several ways to do the core rotation. One based on IF-THEN decision, others based on applying the sign logic. The different versions use the conditional compilation directive of the latest Stamp IDE, and the selection directive, #SELECT method = 1 (enter the number of the method you want to try).

There is a DEBUG line commented out within the core CORDIC loop. If you put that line back in the program, you can better visualize the internal operation of the CORDIC machinery as it rotates through smaller and smaller angles.

more or less precision, further developments

[top](#)

Things will become only a little more difficult when we move to double precision. But not too much so. It is still just shifts, no multiply or divide. We need to be able to compare magnitudes, and add and subtract. And it requires a double precision table of the arctangents and a new larger scaling for unity and 360 degrees.

[<top>](#) [<index>](#) [<home>](#)  [<mailto:info@emesystems.com>](mailto:info@emesystems.com)