

Fixed-Point Trigonometry with CORDIC Iterations

Ken Turkowski, Apple Computer

Introduction to the CORDIC Technique

CORDIC is an acronym that stands for **CO**ordinate **R**otation **D**igital **C**omputer, and was coined by Volder [Volder 59]. Its concepts have been further developed to include calculation of the Discrete Fourier Transform [Despain 74], exponential, logarithm, forward and inverse circular and hyperbolic functions, ratios and square roots [Chen 72] [Walther 71], and has been applied to the anti-aliasing of lines and polygons [Turkowski 82].

It is an iterative fixed-point technique that achieves approximately one more bit of accuracy with each iteration. In spite of merely linear convergence, the inner loop is very simple, with arithmetic that consists of only shifts and adds, so it is competitive with (and even outperforms) floating-point techniques with quadratic convergence, for the accuracy typically required for 2-dimensional raster graphics.

CORDIC Vector Rotation

To rotate a vector $[x \ y]$ through an angle θ , we perform the linear transformation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \quad (\text{eq. 1})$$

If this transformation is accomplished by a sequence of rotations θ_i such that

$$\theta = \sum_i \theta_i \quad (\text{eq. 2})$$

we then have

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \prod_i \begin{bmatrix} \cos \theta_i & \sin \theta_i \\ -\sin \theta_i & \cos \theta_i \end{bmatrix} \quad (\text{eq. 3})$$

where the product is performed on the right with increasing i . We next factor out a $\cos \theta_i$ from the matrix:

$$\begin{aligned} \begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} x \\ y \end{bmatrix} \prod_i \begin{bmatrix} \cos \theta_i & \tan \theta_i \\ -\tan \theta_i & 1 \end{bmatrix} \\ &= \prod_j \cos \theta_j \begin{bmatrix} x \\ y \end{bmatrix} \prod_i \begin{bmatrix} 1 & \tan \theta_i \\ -\tan \theta_i & 1 \end{bmatrix} \end{aligned} \quad (\text{eq. 4})$$

then we constrain the θ_i 's such that

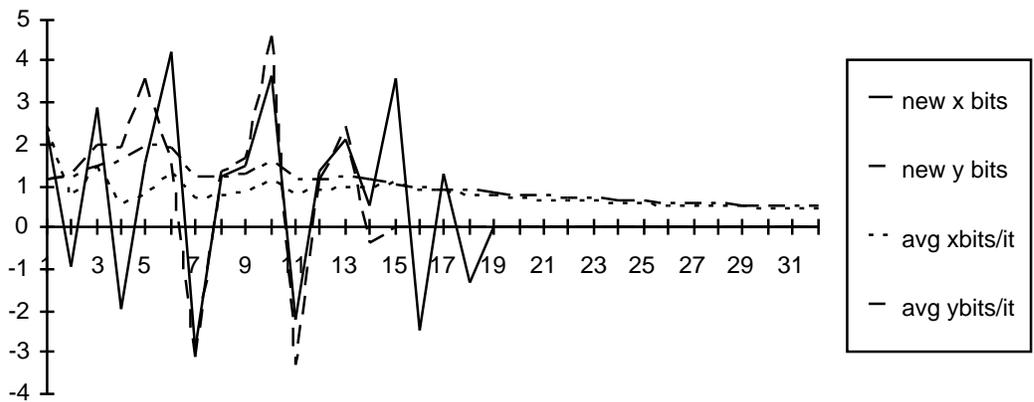
$$\tan \theta_i = \pm 2^{-i} \quad (\text{eq. 5})$$

where the sign is chosen for each i so that (eq. 2) is fulfilled to the desired accuracy. We arrive at:

$$\begin{aligned}
 \begin{bmatrix} x \\ y \end{bmatrix} &= \prod_j \begin{bmatrix} \cos(\tan^{-1} 2^{-j}) & 1 \\ \mp_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\
 &= \frac{1}{\sqrt{1+2^{-2j}}} \prod_j \begin{bmatrix} 1 & \pm 2^{-i} \\ \mp_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
 \end{aligned}
 \tag{eq. 6}$$

This is even simpler than it looks. The factor in braces is a constant for a fixed number of iterations, so it may be precomputed. The matrix multiplications are nothing more than shifts and adds (or subtracts).

The convergence of this product is guaranteed in an interval greater than -90° to 90° when i starts out at 0, although it converges faster when i begins at -1 (i.e. the first shift is upward, the rest downward). Typical convergence is illustrated in the graph below using approximately 16 bit registers. Note that no new bits of accuracy are added after about 16 iterations, but until that point, approximately 1 bit of accuracy is added per iteration.



The scale factor converges 2 bits per iteration and is accurate to 16 bits after 9 iterations, so the limit value could be safely used for all practical numbers of iterations, and has the value 0.2715717684432241 when i starts at -1, and the value 0.6072529350088813 when i starts at 0.

The recommended algorithm is then:

$$\begin{bmatrix} x \\ y \end{bmatrix} = 0.27157177 \prod_{i=-1}^{N-2} \begin{bmatrix} 1 & \pm 2^{-i} \\ \mp_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}
 \tag{eq. 7}$$

or in words:

Start with the largest subangle and either add it or subtract it in such a way as to bring the desired angle closer to zero, and perform the matrix multiplication corresponding to that subrotation with shifts and adds (or subtracts). Continue on in the manner for the specified number of iterations. Multiply the result by the constant scale factor.

Note that 2 extra bits are needed at the high end due to the scale factor. It is difficult to assess the error due to finite word length, but 32 bit registers work well in practice. Expect to have about 2-3 LSB's of noise, even if 32 iterations are performed.

If the original vector was $[r \ 0]$, then rotation by θ performs a polar-to-rectangular conversion. If r is 1, it generates sines and cosines. Rectangular-to-polar conversion can be accomplished by determining the sense of the rotation by the sign of y at each step.

Appendix: Sample C Code

```

# define COSCALE 0x22C2DD1C /* 0.271572 */
# define QUARTER ((int)(3.141592654 / 2.0 * (1 << 28)))
static long arctantab[32] = {          /* MS 4 integral bits for radians */
    297197971, 210828714, 124459457, 65760959, 33381290, 16755422, 8385879,
    4193963, 2097109, 1048571, 524287, 262144, 131072, 65536, 32768, 16384,
    8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 0, 0,
};

CordicRotate(px, py, theta)
long *px, *py;
register long theta; /* Assume that abs(theta) <= pi */
{
    register int i;
    register long x = *px, y = *py, xtemp;
    register long *arctanptr = arctantab;

    /* The -1 may need to be pulled out and done as a left shift */
    for (i = -1; i <= 28; i++) {
        if (theta < 0) {
            xtemp = x + (y >> i);          /* Left shift when i = -1 */
            y      = y - (x >> i);
            x = xtemp;
            theta += *arctanptr++;
        } else {
            xtemp = x - (y >> i);
            y      = y + (x >> i);
            x = xtemp;
            theta -= *arctanptr++;
        }
    }
    *px = frmul(x, COSCALE); /* Compensate for CORDIC enlargement */
    *py = frmul(y, COSCALE); /* frmul(a,b)=(a*b)>>31, high part of 64-bit product */
}

CordicPolarize(argx, argy)
long *argx, *argy; /* We assume these are already in the right half plane */
{
    register long theta, yi, i;
    register long x = *argx, y = *argy;
    register long *arctanptr = arctantab;

```

```

for (i = -1; i <= 28; i++) {
    if (y < 0) { /* Rotate positive */
        yi = y + (x >> i);
        x = x - (y >> i);
        y = yi;
        theta -= *arctanptr++;
    } else { /* Rotate negative */
        yi = y - (x >> i);
        x = x + (y >> i);
        y = yi;
        theta += *arctanptr++;
    }
}
*argx = frmul(x, COSCALE);
*argy = theta;
}

```

References

- Chen 72 Chen, T.C. Automatic computation of exponentials, logarithms, ratios and square roots. *IBM J. Res. Dev.* (July 1972), 380-388.
- Despain 74 Despain, A.M. Fourier transform computers using CORDIC iterations. *IEEE Trans. Comput.*, C-23, 10 (Oct.1974), 993-1001.
- Turkowski 82 Turkowski, K.E. Anti-aliasing through the use of coordinate transformations. *ACM Trans. Graphics*, 1, 3 (July 1982), 215-234.
- Volder 59 Volder, J.E. The CORDIC trigonometric computing technique. *IRE Trans. Electron. Comput.*, EC-8, 3 (Sept. 1959), 330-334.
- Walther 71 Walther, J.S. A unified algorithm for elementary functions. In *Proceedings of AFIPS 1971 Spring Joint Computer Conference*, vol. 38, AFIPS Press, Arlington, Va., 1971, pp. 379-385.