Digital Signal Processing: A Computer Science Perspective
Jonathan Y. Stein

Print ISBN 0-471-29546-9 Online ISBN 0-471-20059-X

16 Function Evaluation Algorithms

Commercially available DSP processors are designed to efficiently implement FIR, IIR, and FFT computations, but most neglect to provide facilities for other desirable functions, such as square roots and trigonometric functions. The software libraries that come with such chips *do* include such functions, but one often finds these general-purpose functions to be unsuitable for the application at hand. Thus the DSP programmer is compelled to enter the field of numerical approximation of elementary functions. This field boasts a vast literature, but only relatively little of it is directly applicable to DSP applications.

As a simple but important example, consider a complex mixer of the type used to shift a signal in frequency (see Section 8.5). For every sample time t_n we must generate both $\sin(\omega t_n)$ and $\cos(\omega t_n)$, which is difficult using the rather limited instruction set of a DSP processor. Lack of accuracy in the calculations will cause phase instabilities in the mixed signal, while loss of precision will cause its frequency to drift. Accurate values can be quickly retrieved from lookup tables, but such tables require large amounts of memory and the values can only be stored for specific arguments. General purpose approximations tend to be inefficient to implement on DSPs and may introduce intolerable inaccuracy.

In this chapter we will specifically discuss sine and cosine generation, as well as rectangular to polar conversion (needed for demodulation), and the computation of arctangent, square roots, Puthagorean addition and logarithms. In the last section we introduce the CORDIC family of algorithms, and demonstrate its applicability to a variety of computational tasks. The basic CORDIC iteration delivers a bit of accuracy, yet uses only additions and shifts and so can be implemented efficiently in hardware.

16.1 Sine and Cosine Generation

In DSP applications, one must often find $\sin(\omega t)$ where the time t is quantized $t = k\,t_s$ and $f_s = \frac{1}{t_s}$ is the sampling frequency.

$$\sin(\omega t) = \sin(2\pi\, f\, k\, t_s) = \sin\left(2\pi \frac{f}{f_s} k\right)$$

The digital frequency of the sine wave, f/f_s, is required to have resolution $\frac{1}{N}$, which means that the physical frequency is quantized to $f = \frac{m}{N} f_s$. Thus the functions to be calculated are all of the following form:

$$\sin\left(2\pi \frac{m}{N} k\right) = \sin\left(\frac{2\pi}{N} i\right) \qquad i \equiv mk = 0 \ldots N$$

In a demanding audio application, $f_s \approx 50$ KHz and we may want the resolution to be no coarser than 0.1 Hz; thus about $N = 500,000$ different function values are required. Table lookup is impractical for such an application.

The best known method for approximating the trigonometric functions is via the Taylor expansions

$$\begin{aligned} \sin(x) &= x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots \\ \cos(x) &= 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots \end{aligned} \tag{16.1}$$

which converge rather slowly. For any given place of truncation, we can improve the approximation (that is, reduce the error made) by slightly changing the coefficients of the expansion. Tables of such corrected coefficients are available in the literature. There are also techniques for actually speeding up the convergence of these polynomial expansions, as well as alternative rational approximations. These approximations tend to be difficult to implement on DSP processors, although (using Horner's rule) polynomial calculation can be pipelined on MAC machines.

For the special case (prevalent in DSP) of equally spaced samples of a sinusoidal oscillator of fixed frequency, several other techniques are possible. One technique that we studied in Section 6.11 exploits the fact that sinusoidal oscillations are solutions of second-order differential or difference equations, and thus a new sine value may be calculated recursively based on two previous values. Thus one need only precompute two initial values and thereafter churn out sine values. The problem with any recursive method of this sort is *error accumulation*. Our computations only have finite accuracy, and with time the computation error builds up. This error accumulation

leads to long-term instability. We can combine recursive computation with occasional nonrecursive (and perhaps more expensive) calculations, but then one must ensure that no sudden changes occur at the boundaries.

Another simple technique that recursively generates sinusoids can simultaneously produce both the sine and the cosine of the same argument. The idea is to use the trigonometric sum formulas

$$\begin{aligned} \sin(\omega k) &= \sin\Big(\omega(k-1)\Big) * \cos(\omega) + \cos\Big(\omega(k-1)\Big) * \sin(\omega) \\ \cos(\omega k) &= \cos\Big(\omega(k-1)\Big) * \cos(\omega) - \sin\Big(\omega(k-1)\Big) * \sin(\omega) \end{aligned} \quad (16.2)$$

with known $\sin(\omega)$ and $\cos(\omega)$. Here one initial value of *both* sine and cosine are required, and thereafter only the previous time step must be saved. These recursive techniques are easily implementable on DSPs, but also suffer from error accumulation.

Let's revisit the idea of table lookup. We can reduce the number of values which must be held in such a table by exploiting symmetries of the trigonometric functions. For example, we do not require twice N memory locations in order to simultaneously generate both the sine and cosine of a given argument, due to the connection between sine and cosine in equation (A.22).

We can more drastically reduce the table size by employing the trigonometric sum formula (A.23). To demonstrate the idea, let us assume one wishes to save sine values for all integer degrees from zero to ninety degrees. This would a priori require a table of length 91. However, one could instead save three tables:

1. $\sin(0°)$, $\sin(10°)$, $\sin(20°)$, ... $\sin(90°)$
2. $\sin(0°)$, $\sin(1°)$, $\sin(2°)$, ... $\sin(9°)$
3. $\cos(0°)$, $\cos(1°)$, $\cos(2°)$, ... $\cos(9°)$

and then calculate, for example, $\sin(54°) = \sin(50°)\cos(4°)+\sin(40°)\sin(4°)$. In this simple case we require only 30 memory locations; however, we must perform one division with remainder (in order to find $54° = 50° + 4°$), two multiplications, one addition, and four table lookups to produce the desired result. The *economy* is hardly worthwhile in this simple case; however, for our more demanding applications the effect is more dramatic.

In order to avoid the prohibitively costly division, we can divide the circle into a number of arcs that is a power of two, e.g., $2^{19} = 524{,}288$. Then every i, $0 \leq i \leq 524{,}288$ can be written as $i = j + k$ where $j = 512(i/512)$ (here / is the integer division without remainder) and $k = i \bmod 512$ can be found by shifts. In this case we need to store three tables:

1. Major Sine: $\sin(\frac{2\pi}{N} 512\, j)$ 512 values
2. Minor Sine: $\sin(\frac{2\pi}{N} k)$ 512 values
3. Minor Cosine: $\cos(\frac{2\pi}{N} k)$ 512 values

which altogether amounts to only 1536 values (for 32-bit words this is 6144 bytes), considerably less than the 524288 values in the straightforward table.

An alternate technique utilizing the CORDIC algorithm will be presented in Section 16.5.

EXERCISES

16.1.1 Evaluate equation (16.2), successively generating further sine and cosine values (use single precision). Compare these values with those returned by the built-in functions. What happens to the error?

16.1.2 Try to find limitations or problems with the trigonometric functions as supplied by your compiler's library. Can you guess what algorithm is used?

16.1.3 The simple cubic polynomial

$$\frac{4}{\pi^3}\, x \left(\frac{3}{4}\pi^2 - x^2\right)$$

approximates $\sin(x)$ to within 2% over the range $[-\frac{\pi}{2} \ldots \frac{\pi}{2}]$. What are the advantages and disadvantages of using this approximation? How can you bring the error down to less than 1%?

16.1.4 Code the three-table sine and cosine algorithm in your favorite programming language. Preprepare the required tables. Test your code by generating the sine and cosine for all whole-degree values from 0 to 360 and comparing with your library routines.

16.1.5 The signal supplied to a signal processing system turns out to be inverted in spectrum (that is, $f \rightarrow f_s - f$) due to an analog mixer. You are very much worried since you have practically no spare processing power, but suddenly realize the inversion can be carried out with practically no computation. How do you do it?

16.1.6 You are given the task of designing a *mixer-filter*, a device that band-pass filters a narrow bandwidth signal and at the same time translates it from one frequency to another. You must take undesired mixer by-products into account, and should not require designing a filter in real-time. Code your mixer filter using the three-table sine and cosine algorithm. Generate a signal composed of a small number of sines, mix it using the mixer filter, and perform an FFT on the result. Did you get what you expect?

16.2 Arctangent

The floating point arctangent is often required in DSP calculations. Most often this is in the context of a rectangular to polar coördinate transformation, in which case the CORDIC-based algorithm given in Section 16.5 is usually preferable. For other cases simple approximations may be of use.

First one can always reduce the argument range to $0 \leq x \leq 1$, by exploiting the antisymmetry of the function for negative arguments, and the symmetry

$$\tan^{-1}(x) = \frac{\pi}{2} - \tan^{-1}\left(\frac{1}{x}\right)$$

for $x > 1$.

For arguments in this range, we can approximate by using the Taylor expansion around zero.

$$\tan^{-1}(x) = x - \tfrac{1}{3}x^3 + \tfrac{1}{5}x^5 - \tfrac{1}{7}x^7 + \cdots \tag{16.3}$$

As for the sine and cosine functions equations (16.1), the approximation can be improved by slightly changing the coefficients.

EXERCISES

16.2.1 Code the arctangent approximation of equation (16.3), summing up N terms. What is the maximum error as a function of N?

16.2.2 How can improved approximation coefficients be found?

16.2.3 Look up the improved coefficients for expansion up to fifth order. How much better is the improved formula than the straight Taylor expansion? Plot the two approximations and compare their global behavior.

16.2.4 For positive x there is an alternative expansion:

$$\tan^{-1}(x) = \frac{\pi}{4} + a_1 y + a_3 y^3 + a_5 y^5 + \ldots \qquad \text{where } y \equiv \frac{x-1}{x+1}$$

Find the coefficients and compare the accuracy with that of equation (16.3).

16.2.5 Make a phase detector, i.e., a program that inputs a complex exponential $s_n = x_n + \mathrm{i}y_n = Ae^{\mathrm{i}(\omega n + \phi_n)}$, computes, and outputs its instantaneous phase $\phi_n = \tan^{-1}(y_n, x_n) - \omega n$ using one of the arctangent approximations and correcting for the four-quadrant arctangent. How can you find ω? Is the phase always accurately recovered?

16.3 Logarithm

This function is required mainly for logarithmic AM detection, conversion of power ratios and power spectra to decibels, as well as for various musical effects, such as *compression* of guitar sounds. The ear responds to both sound intensities and frequencies in approximately logarithmic fashion, and so logarithmic transformations are used extensively in many perception-based feature extraction methods. Considerable effort has also been devoted to the efficient computation of the natural and decimal logarithms in the non-DSP world.

Due to its compressive nature, the magnitude of the output of the 'log' operation is significantly less than that of the input (for large enough inputs). Thus, relatively large changes in input value may lead to little or no change in the output. This has persuaded many practitioners to use overly simplistic approximations, which may lead to overall system precision degradation.

We can concentrate on base-two logarithms without limiting generality since logarithms of all other bases are simply related.

$$\log_a(x) = \Big(\log_2(a)\Big)^{-1} \log_2(x)$$

If only a single bit of a number's binary representation is set, say the k^{th} one, then the log is simple to calculate—it is simply k. Otherwise the bits following the most significant set bit k contribute a fractional part

$$x = \sum_{i=0}^{k} x_i\, 2^i = 2^k + \sum_{i=1}^{k} x_{k-i}\, 2^{k-i} = 2^k \left(1 + \sum_{i=1}^{k} x_{k-i} 2^{-i}\right) = 2^k\,(1+z)$$

with $0 \le z < 1$. Now $\log_2(x) = k + \log_2(1+z)$ and so $0 \le u = \log_2(1+z) < 1$ as well. Thus to approximate $\log_2(x)$ we can always determine the most significant bit set k, then approximate $u(z)$ (which maps the interval $[0 \ldots 1]$ onto itself), and finally add the results. The various methods differ in the approximation for $u(z)$. The simplest approximation is linear interpolation, which has the additional advantage of requiring no further calculation—just copying the appropriate bits. The maximum error is approximately 10% and can be halved by adding a positive constant to the interpolation since this approximation always underestimates. The next possibility is quadratic approximation, and an eighth-order approximation can provide at least five significant digits.

For an alternate technique using the CORDIC algorithm, see Section 16.5.

EXERCISES

16.3.1 Code the linear interpolation approximation mentioned above and compare its output with your library routine. Where is the maximum error and how much is it?

16.3.2 Use a higher-order approximation (check a good mathematical handbook for the coefficients) and observe the effect on the error.

16.3.3 Before the advent of electronic calculators, scientists and engineers used *slide rules* in order to multiply quickly. How does a slide rule work? What is the principle behind the *circular* slide rule? How does this relate to the algorithm discussed above?

16.4 Square Root and Pythagorean Addition

Although the square root operation $y = \sqrt{x}$ is frequently required in DSP programs, few DSP processors provide it as an instruction. Several have 'square-root seed' instructions that attempt to provide a good starting point for iterative procedures, while for others the storage of tables is required.

The most popular iterative technique is the Newton-Raphson algorithm $y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$, which converges quadratically. This algorithm has an easily remembered interpretation. Start by guessing y. In order to find out how close your guess is check it by calculating $z = \frac{x}{y}$; if $z \approx y$ then you are done. If not, the true square root is somewhere between y and z so their average is a better estimate than either.

Another possible ploy is to use the obvious relationship

$$\sqrt{x} = 2^z \implies z = \tfrac{1}{2}\log_2(x)$$

and apply one of the algorithms of the previous section.

When x can only be in a small interval, polynomial or rational approximations may be of use. For example, when x is confined to the unit interval $0 < x < 1$, the quadratic approximation $y \approx -0.5973x^2 + 1.4043x + 0.1628$ gives a fair approximation (with error less than about 0.03, except near zero).

More often than not, the square root is needed as part of a 'Pythagorean addition'.

$$x \oplus y \equiv \sqrt{x^2 + y^2}$$

This operation is so important that it is a primitive in some computer languages and has been the study of much approximation work. For example, it is well known that

$$x \oplus y \approx \text{abmax}(x, y) + k\ \text{abmin}(x, y)$$

with abmax (abmin) returning the argument with larger (smaller) *absolute* value. This approximation is good when $0.25 \leq k \leq 0.31$, with $k = 0.267304$ giving exact mean and $k = 0.300585$ minimum variance.

The straightforward method of calculating $x \oplus y$ requires two multiplications, an addition, and a square root. Even if a square root instruction is available, one may not want to use this procedure since the squaring operations may underflow or overflow even when the inputs and output are well within the range of the DSP's floating point word.

Several techniques have been suggested, the simplest perhaps being that of Moler and Morrison. In this algorithm x and y are altered by transformations that keep $x \oplus y$ invariant while increasing x and decreasing y. When negligible, x contains the desired output.

In pseudocode form:

$p \leftarrow \max(|x|, |y|)$
$q \leftarrow \min(|x|, |y|)$
`while` $q > 0$
    $r \leftarrow (\frac{q}{p})^2$
    $s \leftarrow \frac{r}{4+r}$
    $p \leftarrow p + 2 \cdot s \cdot p$
    $q \leftarrow s \cdot p$
`output` p

An alternate technique for calculating the Pythagorean sum, along with the arctangent, is provided by the CORDIC algorithm presented next.

EXERCISES

16.4.1 Practice finding square roots in your head using Newton-Raphson.

16.4.2 Code Moler and Morrison's algorithm for the Pythagorean sum. How many iterations does it require to obtain a given accuracy?

16.4.3 Devise examples where straightforward evaluation of the Pythagorean sum overflows. Now find cases where underflow occurs. Test Moler and Morrison's algorithm on these cases.

16.4.4 Can Moler-Morrison be generalized to compute $\sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots}$?

16.4.5 Make an amplitude detector, i.e., a program that inputs a complex exponential $s(t) = x(t) + iy(t) = A(t)e^{i\omega t}$ and outputs its amplitude $A(t) = \sqrt{x^2(t) + y^2(t)}$. Use Moler and Morrison's algorithm.

16.5 CORDIC Algorithms

The **CO**ordinate **R**otation for **DI**gital **C**omputers (CORDIC) algorithm is an iterative method for calculating elementary functions using only addition and binary shift operations. This elegant and efficient algorithm is not new, having been described by Volder in 1959 (he applied it in building a digital airborne navigation computer), refined mathematically by Walther and used in the first scientific hand-held calculator (the HP-35), and is presently widely used in numeric coprocessors and special-purpose CORDIC chips.

Various implementations of the same basic algorithmic architecture lead to the calculation of:

- the pair of functions $\sin(\theta)$ and $\cos(\theta)$,
- the pair of functions $\sqrt{x^2 + y^2}$ and $\tan^{-1}(y/x)$,
- the pair of functions $\sinh(\theta)$ and $\cosh(\theta)$,
- the pair of functions $\sqrt{x^2 - y^2}$ and $\tanh^{-1}(y/x)$,
- the pair of functions $\sqrt{a}$ and $\ln(a)$, and
- the function e^a.

In addition, CORDIC-like architectures can aid in the computation of FFT, eigenvalues and singular values, filtering, and many other DSP tasks. The iterative step, the binary shift and add, is implemented in *CORDIC processors* as a basic instruction, analogously to the MAC instruction in DSP processors.

We first deal with the most important special case, the calculation of $\sin(\theta)$ and $\cos(\theta)$. It is well known that a column vector is rotated through an angle θ by premultiplying it by the orthogonal rotation matrix.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}(\theta) \begin{pmatrix} x \\ y \end{pmatrix} \tag{16.4}$$

$$\mathbf{R}(\theta) \equiv \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} = \cos(\theta) \begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix}$$

If one knows numerically the $\mathbf{R}$ matrix for some angle, the desired functions are easily obtained by rotating the unit vector along the x direction.

$$\mathbf{R}(\theta)\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}\cos(\theta)\\\sin(\theta)\end{pmatrix} \tag{16.5}$$

However, how can we obtain the rotation matrix without knowing the values of $\sin(\theta)$ and $\cos(\theta)$? We can exploit the sum rule for rotation matrices:

$$\mathbf{R}\left(\sum_{i=0}^{n}\alpha_i\right)=\prod_{i=0}^{n}\mathbf{R}(\alpha_i) \tag{16.6}$$

and so for $\theta=\sum_{i=0}^{n}\alpha_i$, using equation (16.4), we find:

$$\begin{aligned}\mathbf{R}(\theta) &= \prod_{i=0}^{n}\cos(\alpha_i)\prod_{i=0}^{n}\begin{pmatrix}1 & -\tan(\alpha_i)\\ \tan(\alpha_i) & 1\end{pmatrix}\\ &= \prod_{i=0}^{n}\cos(\alpha_i)\prod_{i=0}^{n}\mathbf{M}_i\end{aligned} \tag{16.7}$$

If we chose the partial angles α_i wisely, we may be able to simplify the arithmetic.

For example, let us consider the angle θ that can be written as the sum of α_i such that $\tan(\alpha_i)=2^{-i}$. Then the $\mathbf{M}$ matrices in (16.7) are of the very simple form

$$\mathbf{M}_i=\begin{pmatrix}1 & -\frac{1}{2^i}\\ \frac{1}{2^i} & 1\end{pmatrix}$$

and the matrix products can be performed using only right shifts. We can easily generalize this result to angles θ that can be written as sums of $\alpha_i=\pm\tan^{-1}(2^{-i})$. Due to the symmetry $\cos(-\alpha)=\cos(\alpha)$, the product of cosines is unchanged, and the $\mathbf{M}$ matrices are either the same as those given above, or have the signs reversed. In either case the products can be performed by shifts and possibly sign reversals. Now for the surprise—one can show that *any* angle θ inside a certain *region of convergence* can be expressed as an infinite sum of $\pm\alpha_i=\pm\tan^{-1}(2^{-i})$! The region of convergence turns out to be $0\le\theta\le 1.7433$ radians $\approx 99.9°$, conveniently containing the first quadrant. Thus for any angle θ in the first quadrant, we can calculate $\sin(\theta)$ and $\cos(\theta)$ in the following fashion. First we express θ as the appropriate sum of α_i. We then calculate the product of $\mathbf{M}$ matrices using only shift operations. Next we multiply the product matrix by the universal constant $K\equiv\prod_{i=0}^{\infty}\cos(\alpha_i)\approx 0.607$. Finally, we multiply this matrix by the unit

column vector in the x direction. Of course, we must actually truncate the sum of α_i to some finite number of terms, but the quantization error is not large since each successive $\mathbf{M}$ matrix adds one bit of accuracy.

Now let's make the method more systematic. In the 'forward rotation' mode of CORDIC we start with a vector along the x axis and rotate it through a sequence of progressively smaller predetermined angles until it makes an angle θ with the x axis. Then its x and y coördinates are proportional to the desired functions. Unfortunately, the 'rotations' we must perform are not pure rotations since they destroy the normalization; were we to start with a unit vector we would need to rescale the result by K at the end. This multiplication may be more costly than all the iterations performed, so we economize by starting with a vector of length K. Assuming we desire b bits of precision we need to perform b iterations in all. We can discover the proper expansion of θ by greedily driving the residual angle to zero. We demonstrate the technique in the following pseudocode:

$x \leftarrow K$
$y \leftarrow 0$
$z \leftarrow \theta$
for $i \leftarrow 0$ **to** $b-1$
  $s \leftarrow \mathbf{sgn}(z)$
  $x \leftarrow x - s \cdot y \cdot 2^{-i}$
  $y \leftarrow y + s \cdot x \cdot 2^{-i}$
  $z \leftarrow z - s \cdot \tan^{-1}(2^{-i})$
$\cos(\theta) \leftarrow x$
$\sin(\theta) \leftarrow y$
error $\leftarrow z$

Of course only additions, subtractions, and right shifts are utilized, and the b values $\tan^{-1}(2^{-i})$ are precomputed and stored in a table. Beware that in the loop the two values x and y are to be calculated simultaneously. Thus to code this in a high-level language place the snippet

for $i \leftarrow 0$ **to** $b-1$
  $\xi \leftarrow x$
  $x \leftarrow \xi - s \cdot y \cdot 2^{-i}$
  $y \leftarrow y + s \cdot \xi \cdot 2^{-i}$

into your code.

Did you understand how θ was decomposed into the sum of the α_i angles? First we rotated counterclockwise by the largest possible angle,

$\alpha_0 = \tan^{-1} 1 = 45°$. If $\theta > \alpha_0$ then the second rotation is counterclockwise from there by $\alpha_1 = \tan^{-1} \frac{1}{2} \approx 26\frac{1}{2}°$ to $71\frac{1}{2}°$; but if $\theta < \alpha_0$ then the second rotation is clockwise to $18\frac{1}{2}°$. At each iteration the difference between the accumulated angle and the desired angle is stored in z, and we simply rotate in the direction needed to close the gap. After b iterations the accumulated angle approximates the desired one and the residual difference remains in z.

In order to calculate the pair of functions $\sqrt{x^2+y^2}$ and $\tan^{-1}(y/x)$, we use the 'backward rotation' mode of CORDIC. Here we start with a vector (x, y) and rotate back to zero angle by driving the y coördinate to zero. We therefore obtain a vector along the positive x axis, whose length is proportional to the desired square root. The z coördinate accumulates the required arctangent.

The following pseudocode demonstrates the technique:

$x \leftarrow X$
$y \leftarrow Y$
$z \leftarrow 0$
for $i \leftarrow 0$ **to** $b-1$
    $s \leftarrow \mathbf{sgn}(y)$
    $x \leftarrow x + s \cdot y \cdot 2^{-i}$
    $y \leftarrow y - s \cdot x \cdot 2^{-i}$
    $z \leftarrow z + s \cdot \tan^{-1}(2^{-i})$
$\sqrt{X^2+Y^2} \leftarrow K \cdot x$
error $\leftarrow y$
$\tan^{-1}(Y/X) \leftarrow z$

Once again the x and y in the loop are to be computed simultaneously.

As mentioned before, the pseudocodes given above are only valid in the first quadrant, but there are two ways of dealing with full four-quadrant angles. The most obvious is to fold angles back into the first quadrant and correct the resulting sine and cosines using trigonometric identities. When the input is x, y and $-\pi < \theta \leq \pi$ is desired, a convenient method to convert CORDIC's z is to use $\theta = a + q * z$ where $q = \mathrm{sgn}(x)\mathrm{sgn}(y)$ and $a = 0$ if $x > 0$, while otherwise $a = \mathrm{sgn}(y)\pi$.

It is also possible to extend the basic CORDIC region of convergence to the full four quadrants, at the price of adding two addition iterations and changing the value of K. The extended algorithm is initialized with

$$\mathrm{tp}_i \leftarrow \begin{cases} 1 & i \leq 0 \\ 2^{-i} & i \geq 0 \end{cases} \qquad \mathrm{atan}_i \leftarrow \begin{cases} \frac{\pi}{4} & i \leq 0 \\ \tan^{-1}(2^{-i}) & i \geq 0 \end{cases}$$

and $K \leftarrow \frac{\sqrt{2}}{4}\prod_{i=1}^{b}\cos\left(\tan^{-1}(2^{-i})\right)$ and, for example, the backward rotation is now carried out by the following algorithm:

$x \leftarrow X$
$y \leftarrow Y$
$z \leftarrow 0$
for $i \leftarrow -2$ **to** $b-1$
  $s \leftarrow \mathbf{sgn}(y)$
  $x \leftarrow x + s \cdot y \cdot \mathbf{tp}_i$
  $y \leftarrow y - s \cdot x \cdot \mathbf{tp}_i$
  $z \leftarrow z + s \cdot \mathbf{atan}_i$
$\sqrt{X^2+Y^2} \leftarrow K \cdot x$
error $\leftarrow y$
$\tan^{-1}(Y/X) \leftarrow z$

Up to now we have dealt only with circular functions. The basic CORDIC iteration can be generalized to

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & ms_i 2^{-i} \\ -s_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \qquad (16.8)$$
$$z_{i+1} = z_i + s_i t_i$$

where for the circular functions $m = +1$ and $t_i = \tan^{-1}(2^{-i})$, for the hyperbolic functions $m = -1$ and $t_i = \tanh^{-1}(2^{-i})$, and for the linear functions $m = 0$ and $t_i = 2^{-i}$. For the circular and hyperbolic cases one must also renormalize by the constants $K = 1/\prod_{i=0}^{n}\sqrt{1+m2^{-2i}}$. For the hyperbolic case additional iterations are always required.

EXERCISES

16.5.1 Code the forward and backward extended-range CORDIC algorithms. Test them by comparison with library routines on randomly selected problems.

16.5.2 Recode the mixer filter from the exercises of Section 16.1 using CORDIC to generate the complex exponential.

16.5.3 Code a digital receiver that inputs a complex signal $s(t) = A(t)e^{i(\omega t+\phi(t))}$, mixes the signal down to zero frequency $s(t) = A(t)e^{i\phi(t)}$ (using forward CORDIC), and then extracts both the amplitude and phase (using backward CORDIC).

Bibliographical Notes

The reader is referred to the mathematical handbook of Abramowitz and Stegun [1] for properties of functions, and polynomial and rational approximation coefficients. For a basic introduction to numerical techniques I recommend [216].

Techniques for speeding up the convergence of polynomial and rational expansions are discussed in [138].

Generation of sinusoids by recursively evaluating a second-order difference equation is discussed in [58].

Mitchell [120] proposed simple linear interpolation for the evaluation of logarithms, while Marino [158] proposed the quadratic approximation.

Knuth's METAFONT typeface design program (which generates the fonts usually used with TEX and LATEX) is an example of a language that has $\oplus$ as a primitive. Its manual and entire source code are available in book form [134]. The abmax-abmin formula for $\oplus$ was apparently first discussed in [222] but later covered in many sources, e.g., [184]. The Moler and Morrison algorithm was first presented in [175] and was developed for software that evolved into the present MATLAB [90].

The CORDIC algorithm was proposed by Volder [266] in 1959, and refined mathematically by Walther [269]. Its use in the first full-function scientific calculator (the HP-35) is documented in [38]. CORDIC's approximation error is analyzed in [107]. Extending CORDIC to a full four-quadrant technique was proposed by [105], while its use for computation of the inverse trigonometric functions is in [162]. CORDIC-like architectures can aid in the computation of the FFT [51, 52], eigenvalues and singular values [60], and many other DSP tasks [106].