Radian Reduction for Trigonometric Functions

Mary H. Payne and Robert N. Hanek
Digital Equipment Corporation
Hudson, MA 01749

1.0 INTRODUCTION

An accurate reduction poses little difficulty for arguments of a few radians. However for, say, a CRAY1, H format on the VAX, or double extended in the proposed IEEE standard, the maximum argument which might be presented for reduction is of the order of 2^16000 radians. Accurate reduction of such an argument would require storage of π (or its reciprocal) to over 16,000 bits. Direct reduction by division (or multiplication) then requires generation of a somewhat larger number of bits in the result in order to guarantee the accuracy of the reduction. Of these bits only the low few bits of the integer part of the quotient (product) and enough bits to correctly round the remainder are relevant; the rest will be discarded.

We present a reduction procedure which has the following properties:

1. For large arguments, it avoids the calculation of most of those bits which will get discarded anyway.

2. With some penalty in speed, it allows efficient calculation of additional bits to maintain full accuracy in the neighborhoods of the zeros of the desired function.

3. Otherwise it has a speed which is nearly independent of the size of the argument.

We believe that the approach is new and represents a significant improvement over procedures currently in use. A variant of the procedure is used in Version 3 of the VAX Math Library.

2.0 HEURISTIC DESCRIPTION OF THE PROCEDURE

For simplicity of exposition we assume a positive argument. The extension to negative arguments is immediate because of the simple trigonometric identities relating positive and negative arguments.

We consider a unified approach to reduction relative to $r = (2*\pi)*2^{(-i)}$ where $i = 0, 1, 2,$ and 3 correspond to full period, half period, quadrant and octant reduction, respectively. Our goal is to determine a non-negative integer J, an integer I with $0 \leq I < 2^i$, and a fraction h with $0 \leq h < 1$ such that

$$x = r*[J*(2^i) + I + h] \tag{1}$$

That is we separate the quotient x/r into integer and fraction parts, with the integer part further split into the integer J with i trailing 0's and the integer $I < 2^i$. Note that the term involving J is a multiple of $2*\pi$, and hence does not contribute to either sin(x) or cos(x).

To evaluate sin(x), we must consider two cases:

1. For I even we write

$$\sin(x) = \sin(I*r)*\cos(r*h) + \cos(I*r)*\sin(r*h)$$

where I*r will be a (possibly zero) multiple of $\pi/2$, so that either its sine or its cosine is zero and the other is ± 1. Hence

$$\sin(x) = \pm\cos(r*h) \text{ or } \pm \sin(r*h)$$

2. For I odd we write

$$\sin(x) = \sin[(I+1)*r]*\cos[r*(1-h)]$$
$$- \cos[(I+1)*r]*\sin[r*(1-h)]$$

Since I+1 is even the above remarks imply

$$\sin(x) = \pm\cos[(1-h)*r] \text{ or } \pm\sin[(1-h)*r]$$

In either case the appropriate choice of sine or cosine and the sign is a function of the "selector bits" contained in I. Similar comments apply to cos(x), and we conclude that knowledge of I and h is sufficient to calculate any of the trigonometric functions.

The above discussion assumes the existence of evaluation procedures for both the sine and cosine functions. If one implements octant reduction (i = 3), both procedures must in fact be available. We consider octant reduction to be the method of choice (and implemented it on VAX) because it costs no more in performance than the other reductions and provides for a higher performance evaluation stage because of the smaller size of the reduced argument.

We recognize, however, that the more traditional method is to use quadrant reduction in such a way that it is always the sine function which is used for the final evaluation. It is not difficult to modify the procedure developed in the following sections to do this. The modifications are described at the end of Section 3.0.

2.1 The Reduction Process

In the following we shall assume that x is a floating point quantity given to p bits of precision and write it in the form

$$x = (2^k)*f \quad \text{with} \quad 1/2 \leq f < 1 \tag{2}$$

To return the function value accurate to p bits we shall assume it is desirable to calculate the normalized value of the equivalent argument, h or (1-h), to an accuracy of (p+K) bits. The number of guard bits, K, is determined by the algorithm used for the final function evaluation and how close to "1/2 ULP" we wish the final accuracy to be.

To determine I and h we use, in effect, a multiplication of x by $1/(2*\pi)$ instead of a division by $2*\pi$. Consequently we shall store a value of $1/(2*\pi)$ to an accuracy sufficient for the calculation. We discuss in Section 3.1 the number of bits which must be stored, and in Section 4 how we calculated these bits.

From Eq. (1), we see that we can write

$$I + h \approx x/r \pmod{2^{i}}$$

where we use the symbol $\approx$ to denote the congruence relationship in modular arithmetic. Substituting from Eq. (2) for x, we have

$$\begin{aligned} I + h &\approx x/r \\ &\approx [(2^{k})*f]*[(2^{i})/(2*\pi)] \pmod{2^{i}} \end{aligned}$$

or, regrouping the factors,

$$\begin{aligned} I + h &\approx (2^{i})*[(2^{p})*f]*\{[2^{(k-p)}]*[1/(2*\pi)]\} \\ &\quad \pmod{2^{i}} \end{aligned}$$

In the above equation note that $F = (2^{p})*f$ is an integer, and that

$$G = [2^{(k-p)}]*[1/(2*\pi)]$$

is readily obtained by shifting the binary point of $1/(2*\pi)$ to the right or left by $|k-p|$ places for $(k-p)$ positive or negative, respectively. Note that a left shift makes no problem if p leading 0's are stored before the binary point of $1/(2*\pi)$. This is so because if $k < 0$, then $x < 1$ and reduction is either unnecessary or more efficiently carried out by subtraction of r from x. Using F and G, we can write

$$I + h \approx F*G*(2^{i}) \pmod{2^{i}}$$

Next we note that G can be split into an integer part J and a (possibly unnormalized) fraction g, with $0 < g < 1$. Then, since J*F is an integer,

$$\begin{aligned} I + h &\approx (2^{i})*(J*F + g*F) \\ &\approx (2^{i})*(g*F) \pmod{2^{i}} \end{aligned}$$

The use of modular arithmetic (mod 2^{i}) is one of the keys of this procedure: It enables us to index into the bit string stored for $1/(2*\pi)$ to find the shifted binary point, discard the integer bits to its left, and work with the fraction bits to its right.

A second key is related to the probable necessity for extended precision arithmetic. Note that even after the integer J is discarded, the i selector bits and at least (p+K) fraction bits are still required for h. This can be achieved without extended arithmetic only if a level of precision higher than (p+K+i) bits is available, something which cannot exist at the highest level of precision.

Because it simplifies both the exposition and the implementation of the procedure, we choose integer arithmetic rather than floating point for the extended precision. Accordingly we let L be the largest number such that the processor can produce exactly the product of two L bit integers. We then express g and F in terms of digits in radix 2^{L}:

$$\begin{aligned} F &= (2^{p})*f \\ &= \mathrm{SUM}\ \{n = 0,(N-1) : F(n)*[2^{(n*L)}]\} \end{aligned}$$

with

$$N = \text{smallest integer} \geq p/L \qquad (3)$$

and

$$g = \mathrm{SUM}\ \{j = 1 \text{ to infinity: } g(j)*[2^{(-j*L)}]\}$$

The F(n) and g(j) are L bit integers, possibly with leading zeros. It follows that we can write

$$\begin{aligned} I + h &\approx (2^{i})*F*g \approx (2^{i})*(J1 + Q) \\ &\approx (2^{i})*Q \pmod{2^{i}} \end{aligned}$$

where J1 is an integer and Q is a radix (2^{L}) fraction:

$$Q = \mathrm{SUM}\ \{j = 1 \text{ to infinity: } Q(j)*[2^{(-j*L)}]\}$$

That is, in implementing the extended precision product F*g, only those partial products contributing to the fractional part, Q, of the product need be generated. Note that the high i bits of Q are promoted to integer bits by the multiplication by 2^{i}, and these are the selector bits.

2.2 Approximating The Equivalent Argument

To proceed further, we must consider a finite approximation Q' of Q obtained by retaining only, say, the high M radix 2^{L} digits in g. As we shall see it may be necessary later to include additional digits of g in its product by F. Therefore we shall express g as the sum of D0, D1, and D2, with

$$D0 = \mathrm{SUM}\ \{j = 1 \text{ to } M\text{: } g(j)*[2^{(-j*L)}]\}$$

$$D1 = g(M+1)*\{2^{[-(M+1)*L]}\}$$

$$\begin{aligned} D2 = \mathrm{SUM}\ \{&m = 2 \text{ to infinity:} \\ &g(M+m)*(2^{[-(M+m)*L]})\} \end{aligned}$$

Using D0 to approximate Q, we obtain

$$\begin{aligned} I' + h' &\approx (2^{i})*F*D0 \approx (2^{i})*(J2 + Q') \\ &\approx (2^{i})*Q' \pmod{2^{i}} \qquad (4) \end{aligned}$$

where Q' is the fractional part of F*D0. For any $M \geq N$ the following properties hold for the various quantities appearing in Eq. (4):

1. Since F is an integer represented in p bits and D0 is a fraction, J2, the integer part of the product, is a p bit integer (perhaps with some leading zero bits).

2. The entire product F*D0 is representable in (M*L+p) bits.

3. Hence its fractional part Q' is representable in M*L bits.

4. Since D0 represents g chopped to M digits, the high M*L bits of F*D0 are accurate except possibly for a carry from the neglected terms in the product. Of these bits p get discarded when J2 is discarded. Hence we shall say that Q' contains (M*L-p) "valid" bits, where, from Eq. (3), (M*L-p) $\geq$ 0.

5. I' = int[(2^i)*Q'] defines the selector bits, accurate except possibly for propagation of a carry from the neglected terms.

6. h' = fract[(2^i)*Q'] contains (M*L-p-i) valid bits. Hence if M is chosen too small h' will contain fewer than (p+K) valid bits.

7. Even if M is large enough to guarantee (p+K) valid bits, h' may have so many leading zero bits that there may be fewer than the (p+K) valid bits desired for the normalized equivalent argument.

8. Similarly h' may have leading bits consisting entirely of ones, and if (1-h) is required for the function evaluation, (1-h') may contain fewer than the desired (p+K) valid bits.

We refer to the last two cases as posing a potential "loss of significance" problem. We know of no way to predict (a priori) either the occurrence or severity of a loss of significance problem. However, it will be easy to test for strings of leading zeros and ones in h' after multiplication of F by D0, if M is sufficiently larger than N.

2.3 Generating Full Accuracy For The Equivalent Argument

We now show how to generate additional terms in the product to get full accuracy for h'. Recall that, from Properties 3 and 4, Q' is representable in M*L bits, of which only the high (M*L-p) are valid. That is Q' = Q1 + Q2, with

Q1 = SUM {j = 1 to (M-N): Q'(j)*[2^(-j*L)]}

Q2 = SUM {j = (M-N+1) to M: Q'(j)*[2^(-j*L)]}

Since Eq. (3) implies N*L $\leq$ p, all digits in Q1 and the high (N*L-p) bits of Q2 are valid except possibly for the propagation of a carry from neglected terms in F*g.

We next consider the contribution of D1, the (M+1)st digit of g, which has the value g(M+1)*[2^(-(M+1))*L]. Its product with the leading digit of F has the value F(N-1)*g(M+1)*[2^(N-M-2)*L]. But this is a two digit product, and the higher of these digits has weight 2^(N-M-1)*L which is identical to the weight of the leading digit of Q2. That is the high digits of Q2 and F*D1 will be "aligned" in the addition necessary to bring in its contribution to the product F*g. Note that the high (N*L-p) bits of F in its digit representation are all zeros, and these bits line up with the high (N*L-p) valid bits of Q2.

At this point we have shown how to generate Q' (and hence I' and h') corresponding to D0 defined relative to M+1 rather than M. Recall that the list of properties holds for any M $\geq$ N. Hence we may define a recursion for successively adding in terms from F*D2 by letting M $\leftarrow$ (M+1) and appropriately updating Q1, Q2, D0, D1, and D2. Each additional term, F*(the new D1) can affect (the new) Q1 only by propagation of a carry from the sum of its product with Q2. Moreover, from Properties 1 through 4, Q1 will always contain M valid digits, and Q2 will always contain (M-N) digits, the most significant of which will always be aligned with the leading digit of the product of F by the new D1. Moreover, the leading (N*L-p) bits of Q2 will always be valid.

We note that if loss of significance arises

1. From leading zeros, any propagation of a carry by the addition of subsequent terms will be stopped when it reaches the leading zeros.

2. From leading ones, there are two cases

   (a) A carry propagates into the string of ones, and hence into the low selector bit, thus converting this case into Case 1 above.

   (b) Otherwise, any propagation of a carry by the addition of terms following the final (p+K) bits calculated after the string is broken will be stopped by the zero which breaks the string, if not earlier.

Hence, in all cases, the selector bits will be valid after the fix-up for loss of significance is complete.

It follows from the above that tests are required on F*D0 to detect an initial occurrence of loss of significance. Further tests are required after addition of each subsequent product of F by the "next" digit in 1/(2*π), to determine whether a loss of significance problem is still present. If it is, additional terms from the product must be added in until it disappears. We show in Section 3.1 that we can guarantee that this process will terminate either in a valid value or in underflow for the equivalent argument.

It is tedious, but not difficult, to verify that loss of significance is a real problem only if the current values of I' and h' are such that:

- o Either I' is even and h' has so many leading zeros that it is an inaccurate approximation to h,
- o Or I' is odd and h' has so many leading ones that (1-h') is an inaccurate approximation to (1-h).

This handling of loss of significance by bringing in additional bits of 1/(2*π) is the third key element of the procedure. It guarantees that the accuracy of trigonometric function evaluation will be uniform over the entire range of values of the argument, including those arguments near to the zeros of the functions.

2.4 Choosing The Initial Approximation

Since the absence of a loss of significance problem is likely the "usual" case, we now derive M1, the smallest value of M, such that if no loss of significance problem occurs, there will be enough valid bits in Q' = (F*D0) (mod 2^i) to complete the reduction without adding more terms to the approximation. That is we wish to determine the minimum M so that there will be (p+K) valid bits in h', the fractional part of (2^i)*Q', in the absence of a loss of significance problem. From Property 4 above, h' contains (M*L-p-i) valid bits. Hence we wish to choose M1 to satisfy:

$$M1*L - p - i > p + K$$

or

$$M1 = \text{smallest integer} > (2*p + K + i)/L$$

With this choice of M = M1, M > N and if h' contains a string of leading ones or of leading zeros whose length exceeds (M1*L-2*p-K) a potential loss of significance problem exists, in which case the analysis of the preceding section can be applied.

3.0 OUTLINE OF THE FULL REDUCTION PROCEDURE

The full procedure for reduction, including handling of the loss of significance problem is as follows:

Step 1: Initialize M to M1 and calculate I', Q' = Q1 + Q2, and h' as described in Sections 2.2 and 2.4. Go to Step 2.

Step 2: Check the current approximation h' for a loss of significance problem, in accordance with Section 2.3. If none exists go to Step 4, otherwise go to Step 3.

Step 3: As discussed in Section 2.3, calculate F*D1 and add to Q2, propagating a carry into Q1, if necessary. Then increment M and, using this new M, update:

(a) D0, D1, and D2.

(b) Q', Q1, and Q2 so that Q1 contains the high (M-N) valid digits, and Q2 the remaining N digits of their sum Q'.

(c) I' and h' as the integer and fractional parts of the product (2^i)*Q', respectivel .

Then go to Step 2.

Step 4: If I' is odd, calculate (1-h') and exit.

Although we have no proof, we conjecture that the number of entries into the implicit loop of Step 3 will be small: The maximum number we have encountered in our implementation on VAX is two.

As promised in Section 2.0, we now show how the above procedure must be modified to implement a quadrant reduction in such a way that the final evaluation will be a sine function: Taking i = 2, and consequently r = π/2, Eq. (1) in Section 2.0 gives

$$x = (\pi/2)*[4*J + I + h],$$

where 0 < h < 1 and I and J are non-negative integers with I < 3. It follows that we may write I as

$$I = 2*I1 + I2,$$

where I1 and I2 are either 0 or 1. We define y to be the equivalent argument, i.e. y = h if I is even and y = 1-h if I is odd. Then to compute sin(x), we note that

$$\sin(x) = [(-1)\hat{}I2]*\sin[y*(\pi/2)].$$

so that we can obtain I and y as described above, evaluate sin[y*(π/2)], and negate the result if the high bit of I is set.

To compute cos(x), we note that

$$\cos(x) = \sin(x + \pi/2)$$

and

$$x + \pi/2 = (\pi/2)*[4*J + (I+1) + h].$$

Hence the net effect of adding π/2 to x is to increment the selector bits by 1. It follows that to compute cos(x), one modifies the computation of Q', in Step 1, so that 4*Q' ≈ 4*F*D0 (mod 4) is replaced by

$$4*Q' \approx 4*F*D0 + 1 \pmod 4$$

and proceeds as for sin(x).

3.1 Storage Requirements For 1/(2*π)

The storage requirements consist of three parts:

1. First recall from Section 2.1 that p leading zeros must be stored to the left of the binary point associated with 1/(2*π) in order to permit "left" shifting in the indexing operation.

2. Recall that the right shift required is (k-p), for an unbiased exponent of the argument exceeding p. Let the maximum unbiased exponent for the data type under consideration be EMAX. Then it must be possible to shift the binary point of 1/(2*π) by (EMAX-p) places to the right, and hence (EMAX-p) bits of storage must be provided.

3. Finally, we must store enough bits in 1/(2*π) so that (except for underflow of

the equivalent argument) an accurate value of h' (or (1-h')) can be returned even with the maximum possible shift in its binary point described above. Let j be the number of bits stored to the right of this shifted binary point. Then F*g' will contain j valid bits, Q' will contain (j-p) valid bits, and h' will contain (j-p-i). Suppose now that h' contains S leading zeros (or ones). Then if EMIN (a negative number) is the largest value of the unbiased exponent for which underflow does not occur, underflow will not occur for h' (or (1-h')) if S = -EMIN but will occur if S > -EMIN. Since we wish to return (p+K) valid bits in the absence of underflow, we must require

$$j - p - i = -EMIN + p + K$$

It follows that if j is chosen to be [-EMIN+(2*p)+i+K], loss of significance will produce either underflow or at least (p+K) valid bits in the normalized equivalent argument. Note that this value of j is appropriate whether the processor stores 0 or, as in IEEE, uses denormalized numbers for underflow.

Combining the results of Items 2 and 3, we see that a total of

$$(EMAX - p) + [-EMIN + (2*p) + i + K] = EMAX - EMIN + p + i + K$$

bits must be stored for $1/(2*\pi)$. Note that about half of these bits are needed for accurately processing large arguments and the other half to accurately process such arguments in the neighborhood of the zeros of the sine and cosine functions. For a 32 bit floating point data type with 8 exponent bits and 24 bits of precision, the necessary storage requirements are approximately ten 32 bit words.

A proof of the conjecture at the end of the preceding section, showing that, for typical values of p and L, a loss of significance of at most a few 2^L digits could occur, would reduce the above storage requirement by almost half.

Finally, p leading zeros must be stored, or some equivalent mechanism must be provided for handling a possible left shift of the binary point.

3.2 Implementation On VAX.

A variant of the above algorithm is used to implement an octant reduction (i = 3) routine for each of the four VAX native mode floating point data types. Since VAX H-format has a 15 bit exponent field, this requires storage of approximately 32,000 bits for $1/\pi$. The necessary constant was generated as described in Section 4. All four routines access the same constant.

We require the equivalent argument to have at least six guard bits, corresponding to K = 6. This allows us to compute the values of the sine and cosine functions with an error bound of slightly more than 3/4 ULP over the entire range of floating point arguments. There is no loss of accuracy near the zeros of these functions.

4.0 RECIPROCAL OF π TO ARBITRARY ACCURACY

R. W. Gosper has derived a remarkable series for the evaluation of the reciprocal of π; see References 1 and 2. This is done by a sequence of transformations for accelerating the convergence of a far more mundane series.

Using the notation B(N,K) to denote the binomial coefficient with value

$$B(N,K) = (N!)/[(K!)*(N-K)!]$$

Gosper's formula reads:

$$4/\pi = 5/4 + SUM\ \{N\geq 1:\ [2\hat{}(-12*N+1)]*(42*N+5)*[B(2*N-1,N)]\hat{}3\}$$

The properties of this formula which make it so remarkable are as follows:

1. The k^{th} term of the formula is exactly representable in 6*k bits.

2. The first n terms of the sum can be represented exactly in 12*n bits.

3. The most significant bit of the k^{th} term has weight at most 2^(1-6*k) and hence each successive term increases the number of valid bits in the sum by at least 6.

4. If $12*k < m + 1 \leq 12*(k+1)$, then the m^{th} bit of $4/\pi$ may be computed using only terms beyond the k^{th}.

We used the following algorithm (written in LISP) to calculate $4/\pi$ to over 30,000 bits:

1. Rewrite the formula in the form

   $4/\pi = 5/4 + SUM\ \{N\geq 1:\ A(N)*C(N)*[2\hat{}-K(N)]\}$

2. Initialize: N = 0, A(0) = 5, C(0) = 1, K(0) = 2

3. Calculate A(N+1) = 42 + A(N), which implies A(N) = 42*N + 5.

4. The recursion for C(N) is calculated as follows: Define J and M by

   J*(2^M) = N + 1, with J an odd integer

   Then, in terms of J and M

   C(N+1) = [C(N)/(J^3)]*(2*N+1)^3
   = odd integer

   K(N+1) = K(N) + 9 + 3*M

5. Finally, calculate the $(N+1)^{st}$ term for $4/\pi$ as

   A(N+1)*C(N+1)*[2^[-K(N+1)]]

   and add it into the sum for $4/\pi$.

From Property 3 it is clear that after k iterations the sum will be accurate (except possibly for a carry) to at least 6*k bits. To insure the accuracy of the first 6*k bits, it is sufficient to determine m so that after (k+m) iterations at least one zero appears between the $6*k^{th}$ bit and the $6*(k+m)^{th}$ bit.

We note here without further discussion that it is possible to implement the above algorithm in such a fashion that $4/\pi$ can be obtained to arbitrary precision and that sufficient information may be saved to permit a later calculation of additional bits without restarting from the beginning.

References

1. R. W. Gosper, "Acceleration of Series," March, 1974, Memo #304, Artificial Intelligence Laboratory, M. I. T.

2. R. W. Gosper, "A Calculus of Series Rearrangements," pp. 121 -151 in "Algorithms and Complexity: New Directions and Recent Results," edited by J. F. Traub, Academic Press, 1976.

****

Enhanced Arithmetic for Fortran

****

Robert Paul Corbett
F77 Project
Computer Systems Research Group
Department of Computer Science
University of California
Berkeley, California 94720

Introduction

The 1978 Fortran standard[1] permits mixed-mode expressions, that is, expressions with operands of differing data types. Expressions containing floating point operands of differing precisions pose special problems of interpretation. Consider the statement

```
DX = (3.0 / 7.0) * DY
```

where DX and DY are double precision variables. A naive programmer would likely expect it to set DX to three-sevenths the value of DY accurate to double precision. However, most Fortran processors will evaluate the parenthesized expression using single precision, and so the result will be accurate only to single precision. Many experienced programmers consider such conventions to be right and proper, but only because they have accepted customary usage as law.

Similar problems can arise for integer arithmetic. Fortran compilers for machines with byte-addressing typically provide two or more sizes of integers. For example, compilers for the IBM 370 and the DEC VAX[†]-11/780 provide 2-and 4-byte integers. In those implementations, the size of an integer type is indicated by an asterisk followed by the number of bytes; if no size is specified, the default size of four bytes is assumed. Expressions containing integer operands of differing sizes pose problems of interpretation very similar to those for floating point expressions. For example, in the program fragment

```
INTEGER*2 I, J
INTEGER*4 K

  . . .

I = 20000
J = 20000
K = 10
K = (I + J) * K
```

the size of the integer format used to evaluate the parenthesized expression affects the result of the assignment. If the 2-byte format is used, the addition will cause an integer overflow or, worse, produce a silly result.

1. Evaluation Strategies

The Fortran standard is intentionally vague about certain details concerning arithmetic expressions. In particular, Fortran processors are free to store constants and accumulate intermediate results in formats wider than those used to represent variables in memory. A number of existing compilers take advantage of that freedom. Machines such as the Honeywell 600 and 6000 series, the Prime 750, and the Intel 8087 have arithmetic registers which are wider than the widest format they normally use to store numeric data. Compilers for those machines routinely accumulate intermediate results to greater precision than is used to store values in memory. DEC's VMS[◁] Fortran compiler evaluates and stores real and complex constants in its double precision format if those constants appear in a context which requires a double precision value.

[†]VAX is a registered trademark of Digital Equipment Corporation.

[◁]VMS is a registered trademark of Digital Equipment Corporation.