Argument Reduction for Huge Arguments: Good to the Last Bit

by K. C. Ng

Works in progress, July 13, 1992

1.0 Introduction

It is not uncommon to encounter trigonometric functions of huge argument.Consider a simple spring problem, under no damping conditions, the motion can be described by a simple equation:

$$u = u_0 \cos(\omega t) + v_0 \sin(\omega t)$$

where u is the displacement from the equilibrium position, u_0 is the initial displacement, v_0 is the initial velocity, ω is the natural frequency of the system (ω= sqrt(k/M, where k is the spring constant and M is the mass). Then as the time t goes on toward infinity, the calculation involves a huge argument.

For a given large argument x (in radians), one can ***always*** reduce it to a y in the interval $[-\pi,\pi]$ so that sin(x)=sin(y) and cos(x)=cos(y), because radian trigonometric functions have period 2π. This process is called argument reduction, It is the first step in computing trigonometric functions. In mathematical terminology, y is equal to $x - 2k\pi$ where k is the nearest integer to $x/(2\pi)$.

Although the description of argument reduction is simple, its implementation is a challenge in floating-point arithmetic, especially when x is huge. The main difficulty is in obtaining full accuracy in y. The expression

$$y = x - 2k\pi$$

must be computed in fixed point arithmetic with precision up to the exponent range of x, if one wishes to evaluate trigonometric functions accurately (say to within 1 ULP (unit in the last place, see [ref 7] for a detailed definition).

For example, if $x=10^{200}$, then k is an integer of around 200 digits wide, and more than 200 digits of π after the decimal are required in the computation.

When the argument is large, most early [floating-point] software writers did not attempt to perform argument reduction precisely. Instead they returned junk, or 0.0, or an error message. As a result many users (and some implementors) have formed the impression that obtaining the correct function value for large inputs is simply

impossible. That is why [8] requires an error message be signaled and the result be 0.0 for any trigonometric functions when the argument is huge. That restriction is unnecessary. The correct answer can be computed quite efficiently.

It is often argued that being concerned about large arguments is unnecessary, because sophisticated users simply know better than to compute with large angles. It is our contention that this position is suboptimal, because:

1. It places an unnecessary burden on the user.
2. The consequences of producing incorrect (inaccurate) answers may be catastrophic; many people assume that computers can do arithmetic very well. While numerical analysts know better, not all programmers are numerical analysts, nor should they be.
3. It is a vendors responsibility to provide answers that are as correct as possible.

One approach to argument reduction that has been taken by some computer industry vendors to solve the problem was the introduction of the concept of machine π, i.e., use a finite approximated value of π to replace the π in the process of argument reduction. This approach simplifies the implementation, and preserves trigonometric identities (ref[1]). The problem is that there is no standard choice of the precision of machine π, which means the value of a trigonometric function is machine dependent as well as precision dependent. Current commercial vendors of math co-processors (like the Intel x87 and Motorolla 68882) use a 66 bit π for their trigonometric functions, which is adequate for IEEE double (53 bits) or double extended (64 bits) floating-point arithmetic, but insufficient for quadruple precision (113 bits).

Around 1982, a way of implementing the infinitely precise π argument reduction was found independently by Bob Corbett at UC Berkeley (ref [2]) and the team of Mary H. Payne and Robert N. Hanek at Digital Equipment Corporation (ref [3]). Both had implemented the precise argument reduction on a VAX™, the former wrote the assembly code in the UC Berkeley release 4.3bsd, the latter in the VMS Fortran library.

Although the method has been known for ten years, it is relatively difficult to implement, which has resulted in the technique being relatively obscure. Many contemporary computing systems still deliver a wide variety of results for large

arguments. Some simply return error messages! Below is a table of sin(x) and cos(x) for $x = 10^{22}$ on various current computing systems:

Table 1: sin(x) and cos(x) for $x = 10^{22}$ in radians[a]

$x=10^{22}$	sin (x)	cos(x)	
Mathematical answer	-0.852200849...	0.523214785...	
VAX(VMS, g or h format)	-0.852200849...	0.523214785...	
HP 28S,48SX	-0.852200849...	0.523214785...	
hp 20-s	-0.852200849...	0.523214785...	
SPARC (default)	-0.852200849...	0.523214785...	
SPARC (special)[b]	-0.65365288...	0.75679449...	66 bit π
SPARC (special)[c]	0.87402806...	0.48587544...	53 bit π
hp 25	-0.944145338	-0.329529334	
HP 700	0.0	0.0	
HP 375,425t (4.3BSD)	-0.65365288...	0.75679449...	66 bit π
sun3,NeXt	-0.65365288...	0.75679449...	
hp 15c	0.79310567...	-0.60908405...	13 decimal π
IBM RS/6000 AIX 3005	-0.852200849...	0.523214785...	
IBM 3090/600S-VF AIX 370	0.0	0.0	
IBX C/370	0.0	0.0	
PC:Borland TurboC 2.0	4.67734e-240	5.97245e-287	
PC:Borland TurboPascal	0.0	0.0	
PC:Zortech C++ Demo Com.	-0.462613	0.88656	
PC:Digitalk Smalltalk/V	0.46261304	-0.8865603	
PC:Derive 2.08,150dig.	-0.852200849...	0.523214785...	
PC:Trilogy	0.0	1.0	
PC:Microsoft GWBasic	0.0	1.0	
PC:LaserGo's GoScript v3	0.0	0.0	
Sharp EL5806	-0.090748172	0.42009155	
Stardent 1520	0.874028061	0.485875445	
Stardent 3040	0.0	1.0	
TRS-80 M100	0.0	0.0	
SGI 4D/240GTX	NaN	NaN[d]	
DECstation 3100	NaN	NaN	
casio (e.g. fx-8100)	Error	Error	
SHARP EL-531A	Error	Error	
TI 34,68,95	Error	Error	

a. Disclaimer: The values listed in this table were contributed by various net volunteers (reader of numeric-interest@validgh.com). These are not the official views of the listed computer system vendors, Sun Microsystems, or of anyone else for that matter.
b. Special code was employed to change the value of π to a 66 bit approximation; all other features of the algorithm and environment remained the same.
c. Special code was employed to change the value of π to a 53 bits approximation; all other features of the algorithm and environment remained the same.
d. NaN stands for Not-a-Number, which is used for an undefined result like 0.0/0.0

Note that many computing systems return 0.0 or an error message when x is large. The worst behavior that the author has come across was on the Casio fx-8100. It produces a fatal error computing sin(x) when x > 26.

In this article, a brief description of the method is presented as well as some implementation notes on a portable argument reduction program for IEEE double precision machines (ref[4]).

2.0 Mathematical Background

2.1 Argument Reduction

Instead of reducing the argument to $[-\pi,\pi]$ as mentioned above, it is more common to reduce it to $[-\pi/4,\pi/4]$. Given x, write

$$x = k(\pi/2) + r, \qquad (1)$$

where k is an integer, and $|r| \le \pi/4$. Let n be the last two bits of k, i.e., n = k mod 4, then we have

Table 2:

n	sin(x)	cos(x)	tan(x)
0	sin(r)	cos(r)	sin(r)/cos(r)
1	cos(r)	-sin(r)	-cos(r)/sin(r)
2	-sin(r)	-cos(r)	sin(r)/cos(r)
3	-cos(r)	sin(r)	-cos(r)/sin(r)

The computation of sin(x) is thus replaced by the computation of sin and cos on the primary interval $[-\pi/4,\pi/4]$. In this primary interval, sin and cos can be approximated accurately by polynomial or rational functions. Readers who are interested in such approximations should consult (ref[5]).

2.2 Formulas for k and *r*

Multiplying (1) by the constant $2/\pi$, we have

$$x \cdot (2/\pi) = k + r \cdot (2/\pi). \qquad (3)$$

Since $|r| \le \pi/4$, $r \cdot (2/\pi) \le 0.5$. That is to say, if

$$y = x \cdot (2/\pi) \qquad (4)$$

then

$$k = [y] \qquad (5)$$

where [ ] denotes rounded to nearest integer, and

$$f = y - k \tag{6}$$

$$r = f \cdot (\pi/2) \tag{7}$$

Note that formulas (4) to (7) cannot be used directly in floating-point arithmetic, especially when x has large exponent. The rounding error in $(2/\pi)$ will be magnified tremendously after the subtraction (6), making the subsequent calculation meaningless.

2.3 How small could *f* be?

From here on, we will focus on IEEE double precision arithmetic (53 significant bits) for specificity. In order to compute *f* accurately, we need to carry enough bits of $2/\pi$ in (4) so that y has enough precision after the binary point to guarantee a full 53 *significant* bit of its fraction part *f*. Since π is a transcendental number, it is possible that $x \cdot (2/\pi)$ may be arbitrarily close to an integer (i.e., *f* may be arbitrarily small). If *f* is as tiny as 2^{-1000}, then 1000 more bits of $2/\pi$ must be kept in storage. To keep the number of bits of $2/\pi$ to minimum, we would like to determine *a priori* the lower bound of *f*.

To answer this question, Prof. W. Kahan of UC Berkeley, using the continued fraction related to π, has devised an algorithm to search all floating-point numbers in working precision that are close to a multiple of $\pi/2$ (ref [6]). In 1983, Stuart McDonald, a graduate student working with Prof. Kahan adapted the algorithm into a C program.

By running the Kahan-McDonald program for IEEE double precision arithmetic, we locate the x that is closest to a multiple of $\pi/2$:

$$x = 6381956970095103 \cdot 2^{797}$$

$$= 5.319372648326541416 71...e+255$$

and (4) becomes

$$y = 4 \cdot (\text{some integer}) + 1. + 2.983942503748063...E-19$$

Thus

$$f = 2.983942503748063...E-19$$

$$r = f \cdot \pi/2 = 4.687165924254624....E-19$$

In order words, in IEEE double precision arithmetic, exhaustive search finds a lower bound for f:

$$|f| \geq 2.983942503748063...E-19$$

$$> 2^{-62}. \tag{8}$$

SunPro
A Sun Microsystems, Inc. Business

So in fixed point arithmetic, there are at most 61 leading zeros in *f*.

2.4 How many bits of $2/\pi$ do we need to store?

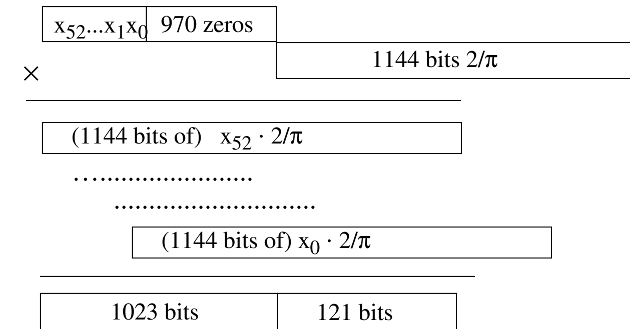
Let's consider a hypothetical worst scenario. Suppose there is an x with maximum exponent 1023 (i.e., $2^{1024} > x \geq 2^{1023}$) such that the fractional part *f* associated with x is the minimum one in (8). In this case, if we want some extra guard bits (say 7 guard bit) for *f* , then y must be accurate to

61(leading zeros) + 53 (non-zero significant bits) + 7 (extra guard bits)

= 121 bits.

Therefore, together with the width of x's exponent, $2/\pi$ must contain

$$1023 + 121 = 1144 \text{ bits.} \tag{9}$$



2.5 Avoid Unnecessary Computation

From 2.1, we need only *f* and the last two bits of *k* for the computation of a trigonometric function. Therefore, one doesn't need to compute the full precision of *k*.

For large x ($x \geq 2^{52}$), let M denote the number of trailing zeros before the binary point (i.e., $x = N \cdot 2^M$ for some odd integer N). We break $2/\pi$ into three pieces

$$2/\pi = A + B + C \tag{10}$$

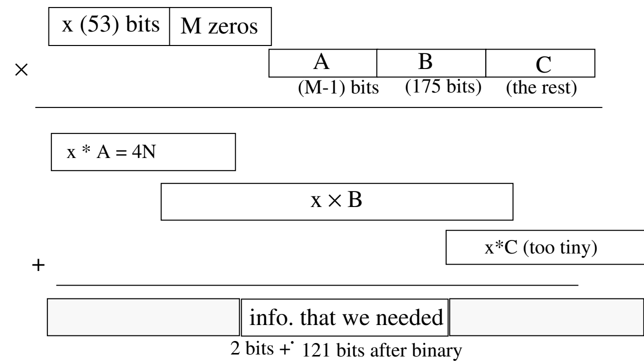
where

A: the first (M-2) bits of $2/\pi$ after binary point,,

B: the $(M-1)^{th}$ bit thru $(M+173)^{th}$ bit of $2/\pi$,

C: from the $(M+174)^{th}$ bit of $2/\pi$ to the rest.

It is easy to see that $x\cdot A = 4N$ for some integer N, which will be discarded, and $|x\cdot C| < 2^{-121}$, which is too tiny to affect the accuracy in f because of (8). Thus in IEEE double arithmetic, $y=x\cdot(2/\pi)$ may be replaced by $y = x\cdot B$



3.0 The computation of x·B

The analysis in section 2.5 took the precaution that y may be close to an integer. In reality, this is rare. One doesn't need that many bits of B in normal situation. Thus one may begin with a B that has fewer bits, say 96 bits, and proceed to compute y, k and *f*. We then check whether *f* suffers cancellation. If it does, then we repeat the computation with more bits in B.

As for the computation of x·B, one can simulate multiprecision arithmetic. There are many ways to accomplish this. We outline one approach below, and leave the details to the reader.

Break up B in several 24-bit pieces. If one begins with a 96 bit B, then there will be four pieces. Denote them to be

$$B = b_1+b_2+b_3+b_4$$

and break up x in three 24-bit pieces:

$$x = x_1+x_2+x_3.$$

Then $x\cdot B =$

$$\begin{array}{rr} & b_1+b_2+b_3+b_4 \\ \times & x_1+x_2+x_3 \end{array}$$

$$\begin{array}{r} \text{---} \\ x_3\cdot b_1 + x_3\cdot b_2 + x_3\cdot b_3 + x_3\cdot b_4 \\ x_2\cdot b1 + x_2\cdot b_2 + x_2\cdot b_3 + x_2\cdot b_4 \quad\quad \\ x_1\cdot b_1 + x_1\cdot b_2 + x_1\cdot b_3 + x_1\cdot b_4 \quad\quad\quad\quad \\ \text{--} \\ y_1 + y_2 + y_3 + y_4 + y_5 + y_6 \end{array}$$

Here y_i's are double precision numbers and are exact, because each product $b_i\cdot x_j$ is only a 48-significant bit number and the sums fit exactly in 50 significant bits.

The final step is to sum up y_i to determine the nearest integer k and the fraction f. Here is one algorithm:

For i from 1 to 6

$y_i = y_i \bmod 4$; (remove unwanted integer parts)

$y = y_1+(y_2+(y_3+(y_4+(y_5+y_6))))$; (in that order)

$t = (((((y_1-y)+y_2)+y_3)+y_4)+y_5)+y_6$;(compensation term for y)

n = [y] rounded to the nearest integer;

$f = (y-n)+t$;

Finally, *r* is computed by $f\cdot(\pi/2)$.

4.0 Implementation on SPARC

A variant of the above algorithm was implemented for SPARC by SunPro, and is incorporated in the libm which ships with SPARCompilers. Single, double and quadruple precision trigonometric functions are included. Loss of significant bits is detected as a special case, and extra bits are used to compute *f* and *r*. The accuracy of sin(x) and cos(x) over the entire range is below one ULP.

Essentially there are three cases. For small arguments, there is no reduction required. For medium size arguments, table driven algorithms which are only a few percent slower than simpler algorithms which provide worse (and in the extreme cases, significantly worse) quality results, are employed. For huge arguments, methods such as described in this paper are employed. Being correct is a virtue in and of itself!

5.0 References

[1] *HP-15C Advanced Functions Handbook*, p.184-186

[2] *BSD 4.3 libm*

[3] M. Payne and R. Hanek, “*Radian Reduction for Trigonometric Functions*”, Signum, p19-24, Jan 1983.

[4] *IEEE Standard754-1985 for Binary Floating-point Arithmetic,* IEEE, (1985). Reprinted in SIGPLAN 22(2) pp9-25.

[5] W. Cody and W. Waite, *Software Manual for the Elementary Function*s, Prentice-Hall, Englewood Cliffs, N.J., 1980.

[6] W. Kahan, “*Minimizing* $q \cdot m - n$”, unpublished research notes, March 1983.

[7] David Goldberg, “*What Every Computer Scientist Should Know About Floating-Point Arithmetic*”, ACM Computing Surveys 23, 1 (March 1991), 5-48. A version of it is reprinted in SunPro’s *Numerical Computation Guide.*

[8] AT&T, *System V Interface Definition*, Third Edition, Vol 1, 1989, pp 7-184.