A new range reduction algorithm

David Defour   Peter Kornerup   Jean-Michel Muller
Nathalie Revol

September 26, 2001

Abstract

Range reduction is a key point for getting accurate elementary function routines. We introduce a new algorithm that is fast for input arguments belonging to the most common domains, yet accurate over the full double precision range.

1 Introduction

The algorithms used for evaluating elementary functions only give correct results if the argument is within a given small interval, usually centered at zero. To evaluate an elementary function $f(x)$ for any x, it is necessary to find some "transformation" that makes it possible to deduce $f(x)$ from some value $g(x^*)$, where

- x^*, called the *reduced argument*, is deduced from x;
- x^* belongs to the convergence domain of the algorithm implemented for the evaluation of g.

In practice, range reduction needs care for the trigonometric functions. With these functions, x^* is equal to $x - kC$, where k is an integer and C an integer multiple of $\pi/4$).

A poor range reduction method may lead to catastrophic accuracy problems when the input arguments are large or close to an integer multiple of C. It is easy to understand why a bad range reduction algorithm gives inaccurate results. The naive method consists of performing the computations

$$\begin{aligned} k &= \left\lfloor \frac{x}{C} \right\rfloor \\ x^* &= x - kC \end{aligned}$$

using the machine precision. When kC is close to x, almost all the accuracy, if not all, is lost when performing the subtraction $x - kC$. For instance, if $C = \pi/2$ and $x = 584664.53$ the correct value of x^* is $-0.0000000016757474710\cdots$, and the corresponding value of k is 372209. Directly computing $x - k\pi/2$ on a calculator with 8-digit arithmetic (assuming rounding to the nearest, and replacing $\pi/2$ by the nearest exactly-representable number), then one gets $-.027247600$. Hence, such a poor range reduction would lead to a computed value of $\cos(x)$ equal to -0.027244229, whereas the correct value is $0.0000000016757474710155235\ldots$

A first solution to overcome the problem consists of using multiple precision arithmetic, but this may make the computation much slower. Moreover, it is not that easy to predict the precision with which the computation should be performed.

Most common input arguments to the trigonometric functions are small (say, less than 8), or sometimes medium (say, between 8 and approximately 2^{60}). They are rarely huge (say, greater than 2^{60}). We want to design methods that are fast for the frequent cases, and accurate for all cases.

First we describe Payne and Hanek's method [5] which provides an accurate range reduction, but has the drawback of being fairly expensive in term of operations. After a short overview for computing the worst case, in the second section we present our algorithm dedicated for small and medium size argument. In the third section we compare our method with some other available methods, which justifies the use of our algorithm for small and medium size argument.

1.1 The Payne and Hanek Reduction Algorithm

We assume in the following that we want to perform range reduction for the trigonometric functions, with $C = \pi/4$, and that the convergence domain of the algorithm used for evaluating the functions contains $I = [0, \pi/4]$. An adaptation to other cases is straightforward.

From an input argument x, we want to find the reduced argument x^* and an integer k, that satisfy:

$$\begin{aligned} k &= \left\lfloor \frac{4}{\pi} x \right\rfloor \\ x^* &= \frac{\pi}{4} \left(\frac{4}{\pi} x - k \right) \end{aligned} \qquad (1)$$

Once x^* is known, it suffices to know $k \bmod 8$ to calculate $\sin(x)$ or $\cos(x)$ from x^*. If x is large, or if x is very close to a multiple of $\pi/4$, the direct use of (1) to determine x^* may require the knowledge of $4/\pi$ with very large precision, and a cost-expensive multiple precision computation if we wish the range reduction to be accurate.

Now let us present Payne and Hanek's reduction method [5, 6]. Assume an n-bit mantissa, radix 2 floating point format (the number of bits n includes the possible hidden bit; for instance, with an IEEE double-precision format, $n = 53$). Let x be the floating-point argument to be reduced and let e be its unbiased exponent, so

$$x = X \times 2^{e-n+1}$$

where X is an n-bit integer satisfying $2^{n-1} \leq X < 2^n$. We can assume $e \geq -1$ (since if $e < -1$, no reduction is necessary). Let

$$\alpha_0 \cdot \alpha_{-1}\alpha_{-2}\alpha_{-3}\alpha_{-4}\alpha_{-5}\cdots$$

be the infinite binary expansion of $\alpha = 4/\pi$, and define an integer parameter p, used to specify the required accuracy of the range reduction. Then rewrite $\alpha = 4/\pi$ as

$$\text{Left}(e,p) \times 2^{n-e+2} + (\text{Medium}(e,p) + \text{Right}(e,p)) \times 2^{-n-e-1-p},$$

where

$$\begin{cases} \text{Left}(e,p) & = \quad \alpha_0\alpha_{-1}\cdots\alpha_{n-e+2} \\ \text{Medium}(e,p) & = \quad \alpha_{n-e+1}\alpha_{n-e}\cdots\alpha_{-n-e-1-p} \\ \text{Right}(e,p) & = \quad 0.\alpha_{-n-e-2-p}\alpha_{-n-e-3-p}\alpha_{-n-e-4-p}\alpha_{-n-e-5-p}\cdots. \end{cases}$$

Figure 1 shows this splitting of the binary expansion of α.

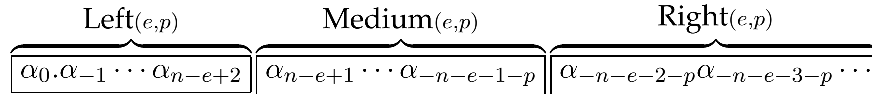

Figure 1: *The splitting of the digits of* $4/\pi$ *in Payne and Hanek's reduction method.*

The basic idea of the Payne-Hanek reduction method is to notice that, if p is large enough, $\text{Medium}(e,p)$ contains the only digits of $\alpha = 4/\pi$ that matter for the range reduction. Since

$$\begin{aligned} \frac{4}{\pi}x \quad = \quad & \text{Left}(e,p) \times X \times 8 + \text{Medium}(e,p) \times X \times 2^{-2n-p} \\ & + \text{Right}(e,p) \times X \times 2^{-2n-p}, \end{aligned}$$

the number Left$(e,p) \times X \times 8$ is a multiple of 8, so that once multiplied by $\pi/4$ (see Eq. (1)), it will have no influence on the trigonometric functions. Right$(e,p) \times X \times 2^{-2n-p}$ is less than 2^{-n-p}; therefore it can be made as small as desired by adequately choosing p.

1.2 How Can We Find Worst Cases for Range Reduction ?

Performing an error analysis for a range reduction algorithm requires the knowledge of the smallest possible reduced argument for all possible inputs in a given format. Computing this value is rather easy, using an algorithm due to Kahan (a C program that implements the method can be found at `http://http.cs.berkeley.edu/~wkahan`. A Maple program is given in [3]). The algorithm uses the continued fraction theory. For instance, a few minutes of calculation suffice to find that the double precision number greater than 8, and less than $2^{63}-1$ which is closest to a multiple of $\pi/2$ is

$$\Gamma = 6411027962775774 \times 2^{-48}.$$

The distance between Γ and the closest multiple of $\pi/2$ is

$$\epsilon \approx 3.9405531196482 \times 10^{-19} > 2^{-62}.$$

So if we apply a range reduction from $[8, 2^{63}-1]$ to $[-\pi/4, \pi/4]$ in double precision arithmetic, we loose at most 61 bits of accuracy.

2 High-radix modular reduction method: New method

As said in the introduction, our general philosophy is that we must give results:

1. always correct, even for rare cases;
2. as quickly as possible for frequent cases.

A way to deal with that is to build a fast algorithm for input arguments with a small exponent, and to use a slower yet still accurate algorithm for input argument with a large exponent.

To do so, in the following we focus on input arguments with a "reasonably small" exponent. More precisely, we assume that the double precision input argument x has absolute value less than $2^{63}-1$. We assume that Payne and Hanek's method will be used for larger arguments. For straightforward symmetry reasons, we can assume that x is positive. We also assume that x is larger than or equal to 8. We then proceed as follows:

1. We define $I(x)$ as x rounded to the nearest integer, and $F(x)$ as $x - I(x)$. Note that $F(x)$ is exactly representable in double precision, and that for $x \geq 2^{52}$, we have $F(x) = 0$ and $I(x) = x$. Also, since $x \geq 8$, the last mantissa bit of $F(x)$ has weight greater than or equal to 2^{-49};

2. $I(x)$ is split into eight 8-bit parts:

$$\begin{cases} I_0(x) \text{ contains bits } 0 \text{ to } 7 \text{ of } I(x) \\ I_1(x) \text{ contains bits } 8 \text{ to } 15 \text{ of } I(x) \\ I_2(x) \text{ contains bits } 16 \text{ to } 23 \text{ of } I(x) \\ I_3(x) \text{ contains bits } 24 \text{ to } 31 \text{ of } I(x) \\ I_4(x) \text{ contains bits } 32 \text{ to } 39 \text{ of } I(x) \\ I_5(x) \text{ contains bits } 40 \text{ to } 47 \text{ of } I(x) \\ I_6(x) \text{ contains bits } 48 \text{ to } 55 \text{ of } I(x) \\ I_7(x) \text{ contains bits } 56 \text{ to } 63 \text{ of } I(x) \end{cases}$$

so that

$$I(x) = I_0(x) + 2^8 I_1(x) + 2^{16} I_2(x) + \ldots + 2^{56} I_7(x).$$

Note that in general there may be an integer $j < 7$ such that $I_i(x) = 0$ for all $i \geq j$.

As mentioned above, our goal is to always provide correct results even for the worst case for which we lose 61 bits of accuracy. Then we need to store $(I_i(x) \bmod^* \pi/2)$ [1] with at least

$$61(\text{leading zeros}) + 53(\text{non-zero significant bits}) + p(\text{extra guard bits}) \\ = 114 + p \text{ bits.}$$

To reach that precision, all the numbers $(I_i(x) \bmod^* \pi/2)$, which belong to $[-1/2, +1/2]$, are stored in tables as the sum of three double precision numbers:

$$\begin{cases} T_{hi}(i,w) \text{ contains bits of weight } 2^{-1} \text{ to } 2^{-49} \text{ of } ((2^{8i}w) \bmod^* \pi/2) \\ T_{med}(i,w) \text{ contains bits of weight } 2^{-50} \text{ to } 2^{-98} \text{ of } ((2^{8i}w) \bmod^* \pi/2) \\ T_{lo}(i,w) \text{ contains bits of weight } 2^{-99} \text{ to } 2^{-147} \text{ of } ((2^{8i}w) \bmod^* \pi/2) \end{cases}$$

where w is an 8-bit integer. Note that $T_{hi}(i,w) = T_{med}(i,w) = T_{lo}(i,w) = 0$ for $w = 0$. The three tables T_{hi}, T_{med} and T_{lo} need 11 address bits. The

[1] defined such that $-\pi/4 \leq (X \bmod^* \pi/2) < \pi/4$

total amount of memory required by these tables is $3 \cdot 2^{11} \cdot 8 = 48$ Kbytes. The largest possible value of

$$\left| \left(\sum_{i=0}^{7} (I_i(x) \bmod^* \pi/2) \right) + F(x) \right|$$

is bounded by $2\pi + \frac{1}{2}$, which is less than 8. From this we deduce that $S_{hi}(x) = \left(\sum_{i=0}^{7} T_{hi}(i, I_i(x)) \right) + F(x)$ is a multiple of 2^{-49} and has absolute value less than 8. *S_{hi} is therefore exactly representable in double-precision floating-point arithmetic* (it is even representable with 52 bits only). Therefore, with a correctly rounded arithmetic (such as the one provided on any system that follows the IEEE-754 standard for floating-point arithmetic), it will be exactly computed.

3. We compute

$$\begin{cases} S_{hi}(x) &= \left(\sum_{i=0}^{7} T_{hi}(i, I_i(x)) \right) + F(x) \\ S_{med}(x) &= \sum_{i=0}^{7} T_{med}(i, I_i(x)) \\ S_{lo}(x) &= \sum_{i=0}^{7} T_{lo}(i, I_i(x)) \end{cases}$$

These three sums are computed exactly in double precision arithmetic, without any rounding error. Observe that these summations can be terminated when starting from $i = 0$ and it is known from some j that $I_i(x) = 0$ for $i \geq j$.

The number $S(x) = S_{hi}(x) + S_{med}(x) + S_{lo}(x)$ is equal to x minus an integer multiple of $\pi/2$ plus a small error (bounded by $8 \times 2^{-148} = 2^{-145}$), due to the fact that the values $(2^{8i}w) \bmod^* \pi/2$ are rounded to the bit of weight 2^{-147}. And yet, $S(x)$ is not the final reduced argument, since its absolute value may be significantly larger than $\pi/4$. We therefore may have to add or subtract a multiple of $\pi/2$ from $S(x)$ to get the final result. Define $C_{hi}(k)$, for $k = 1, 2, 3, 4$, as the multiple of 2^{-49} that is closest to $k\pi/2$. $C_{hi}(k)$ is exactly representable as a double precision number. Define $C_{med}(k)$ as the multiple of 2^{-98} that is closest to $k\pi/2 - C_{hi}(k)$ and $C_{lo}(k)$ as the double precision number that is closest to $k\pi/2 - C_{hi}(k) - C_{med}(k)$.

4. We now proceed as follows:

   - If $|S_{hi}(x)| \leq \pi/4$ then we define

$$\begin{aligned} R_{hi}(x) &= S_{hi}(x) \\ R_{med}(x) &= S_{med}(x) \\ R_{lo}(x) &= S_{lo}(x) \end{aligned}$$

- Else, let k_x be such that $C_{hi}(k_x)$ is closest to $|S_{hi}(x)|$. We successively compute:
  - If $S_{hi}(x) > 0$

$$\begin{array}{lcl} R_{hi}(x) & = & S_{hi}(x) - C_{hi}(k_x) \\ R_{med}(x) & = & S_{med}(x) - C_{med}(k_x) \\ R_{lo}(x) & = & S_{lo}(x) - C_{lo}(k_x) \end{array}$$

  - Else,

$$\begin{array}{lcl} R_{hi}(x) & = & S_{hi}(x) + C_{hi}(k_x) \\ R_{med}(x) & = & S_{med}(x) + C_{med}(k_x) \\ R_{lo}(x) & = & S_{lo}(x) + C_{lo}(k_x) \end{array}$$

  Again, $R_{hi}(x)$ and $R_{med}(x)$ are exactly representable in double precision arithmetic, and, hence, they are exactly computed (for instance, $R_{med}(x)$ has an absolute value less than $2^{-46} + 2^{-49}$, and is a multiple of 2^{-98}). The value $R_{lo}(x)$ is computed with error less than or equal to 2^{-149}.

The number $R(x) = R_{hi}(x) + R_{med}(x) + R_{lo}(x)$ is equal to x minus an integer multiple of $\pi/2$ plus an error bounded by $2^{-145} + 2^{-148}$.

This step is also used (alone, without the previous steps) to reduce small input arguments, less than 8. This allows our algorithm to perform range reduction for both kind of arguments, small and medium size. The reduced argument is now stored as the sum of tree double precision numbers. We now want the reduced argument as the sum of two double precision numbers. To do that, we use the following result:

Theorem 1 (Fast2sum algorithm) [2, page 221, Thm. C] *Let a and b be floating-point numbers, with $|a| \geq |b|$. Assume the used floating-point arithmetic provides correctly rounded results with rounding to the nearest. The following algorithm*

```
fast2sum(a,b):
   s := a + b
   z := s - a
   r := b - z
```

computes two floating-point numbers s and r that satisfy:

- *$r + s = a + b$ exactly;*
- *s is the floating-point number which is closest to $a + b$.*

5. We will get the final result of the range reduction as follows. Let p be the integer parameter that is used to define the required accuracy.

   - If $R_{hi}(x) > 1/2^p$, then we compute

     $$(y_{hi}, y_{lo}) = \text{fast2sum}(R_{hi(x)}, R_{med(x)}).$$

     The 2 floating-point numbers y_{hi} and y_{lo} contain the reduced argument with relative error less than $2^{-96+p} + 2^{-99+p}$;

   - If $R_{hi}(x) = 0$, then we compute

     $$(y_{hi}, y_{lo}) = \text{fast2sum}(R_{med(x)}, R_{lo(x)}).$$

     Since the smallest possible value of the reduced argument is larger than 2^{-62}, the 2 floating-point numbers y_{hi} and y_{lo} contain the reduced argument with *relative* error less than

     $$\frac{2^{-145} + 2^{-148}}{2^{-62}} \approx 2^{-82}$$

   - If $0 < R_{hi(x)} \leq 1/2^p$, then we successively compute $(y_{hi}, temp1) =$ fast2sum$(R_{hi}(x), R_{med}(x))$ and $y_{lo} = R_{lo}(x) + temp1$. The 2 floating-point numbers y_{hi} and y_{lo} contain the reduced argument with *absolute* error less than 2^{-98}.

2.1 The algorithm

We can now sketch the complete algorithm as follows:

Algorithm 1 (Range Reduction)

Stimulus: *A double precision floating point number $x > 0$ and an integer $p > 0$ specifying the required precision in bits.*

Response: *A number y given as the sum of two double precision floating point numbers y_{hi} and y_{lo}, such that $-\pi/4 \leq y < \pi/4$ and $y = x - k\frac{\pi}{2} + \epsilon$ for some integer k, with absolute error $|\epsilon| < 2^{-98}$ for $0 < |y| \leq 2^{-p}$, and relative error $\left|\frac{\epsilon}{y}\right| < 2^{-95+p}$ otherwise.*

Method: **if** $x \geq 2^{63} - 1$ **then**

$\{$*Apply the method of Payne and Hanek.*$\}$

else if $x \leq 8$ **then**

$S_{hi} \leftarrow x$; $S_{med} \leftarrow 0$; $S_{low} \leftarrow 0$;

else

$I \leftarrow \text{round}(x)$; $F \leftarrow x - I$;

$S_{hi} \leftarrow F$; $S_{med} \leftarrow 0$; $S_{low} \leftarrow 0$;

$i \leftarrow 0$;

while $I \neq 0$ **do**

$w \leftarrow I \ mod\ 2^8$;

$S_{hi} \leftarrow S_{hi} + T_{hi}(i, w)$;

$S_{med} \leftarrow S_{med} + T_{med}(i, w)$;

$S_{low} \leftarrow S_{low} + T_{low}(i, w)$;

$I \leftarrow I \text{ div } 2^8$; $i \leftarrow i + 1$;

if $|S_{hi}| \geq \pi/4$ **then**

$k \leftarrow \text{Reduce}(|S_{hi}|)$

$S_{hi} \leftarrow S_{hi} + (-1)^{\text{sign}(\text{S}_{\text{hi}})} C_{hi}(k)$;

$S_{med} \leftarrow S_{med} + (-1)^{\text{sign}(\text{S}_{\text{hi}})} C_{med}(k)$;

$S_{low} \leftarrow S_{low} + (-1)^{\text{sign}(\text{S}_{\text{hi}})} C_{low}(k)$;

if $|S_{hi}| > 2^{-p}$ **then**

$(y_{hi}, y_{lo}) \leftarrow \text{fast2sum}(S_{hi}, S_{med})$;

else if $S_{hi} = 0$ **then**

$(y_{hi}, y_{lo}) \leftarrow \text{fast2sum}(S_{med}, S_{low})$;

else

$(y_{hi}, temp) \leftarrow \text{fast2sum}(S_{hi}, S_{med})$;

$y_{lo} \leftarrow temp + S_{low}$;

Where: *The function Reduce($|S_{hi}|$) chooses the appropriate multiple k of $\pi/2$, represented as the triple $(C_{hi}(k), C_{med}(k), C_{low}(k))$.*

3 Cost of the algorithm

Counting the number N of double precision floating point operations we find for $|x| < 2^{63}$, that $N = 10 + 3\lceil \log_{256} x \rceil$, i.e., $13 \leq N \leq 34$, and the number of table look-ups is $3\lceil \log_{256} x \rceil$.

A variant of our algorithm consists in first computing S_{hi}, S_{med} and R_{hi}, R_{med} only. Then, during the fifth step of the algorithm, if the accuracy does not suffice, compute T_{low} and R_{low}. This slight modification can reduce the number of elementary operations in the (most frequent) cases where no extra accuracy is needed. We can also reduce the table size by 8 Kbytes by storing the T_{low} values in single precision only, instead of using double precision.

In this section we compare our method to other algorithms on the same input range $[8, 2^{63} - 1]$: Payne and Hanek's methods (see Section (1.1)) and the modular range reduction method described in [1]. Concerning Payne and Hanek's method we used the version of the algorithm used by Sun Microsystems [4]. We chose as criteria for the evaluation of the algorithms the table size, the number of table access and the number of floating-point multiplications, divisions and additions.

	# Elementary operations	# Table accesses	Table size in Kbytes
Our algorithm	13/34	3/24	48(40)
Payne & Hanek	55/103	1	0.14
Modular range reduction	150	53	2

The table shows the potential advantages of our algorithm for medium-sized input argument. Payne and Hanek's method over that range doesn't need much memory, but roughly requires three times as many operations. The Modular range reduction has the same characteristics as Payne and Hanek's method concerning the table size needed and the number of elementary operation involved, but requiring more table accesses. Our algorithm is then a good compromise between size table and number of operations for range reduction of medium size argument.

4 Conclusions

We have presented an algorithm for accurate range reduction of input arguments with absolute value less than $2^{63} - 1$. This table-based algorithm gives accurate results for the most frequent cases. In order to cover the whole double precision domain for input arguments, we suggest to use Payne and Hanek's algorithm for huge arguments. A major drawback of our method lies in the table size needed, thus a future effort will be to reduce the table size, while keeping a good tradeoff between speed and accuracy.

References

[1] M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction: A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3):162–175, March 1995.

[2] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, MA, 1973.

[3] J. M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[4] K. C. Ng. Argument reduction for huge arguments: Good to the last bit (can be obtained by sending an e-mail to the author: kwok.ng@eng.sun.com). Technical report, SunPro, 1992.

[5] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[6] R. A. Smith. A continued-fraction analysis of trigonometric argument reduction. *IEEE Transactions on Computers*, 44(11):1348–1351, November 1995.