Improved Complex Division

Michael Baudin (DIGITEO)
Robert Smith (Stanford University)

Version 0.1
Februar 2011

Abstract

This document presents an improved, but simple, complex division. We present a proof of the robustness of the improved algorithm. Numerical simulations show that the algorithm performs well in practice and is significantly more accurate than other known implementations.

Contents

1 Introduction

In floating point arithmetic, designing a robust complex division algorithm is surprisingly difficult. We mainly focus in this article on double precision binary 64 floating point numbers, since these are the floating point numbers used in Scilab.

Indeed, it is easy to present numerical experiments where a naive, straightforward implementation fails to produce an accurate answer. One of the main reason of this kind of failure is the

term c^2+d^2 term which appears in the complex divions expression. This intermediate term might overflow, which is the main reason why the naive method fails to work in practice for a sufficiently large range of input values.

Smith's 1962 method [13] is based on an improved algorithm which greatly improves the robustness of the complex division in floating point arithmetic. Actually, this algorithm is not as robust as we might think. It is easy to find particular complex divisions where Smith's method fails.

Many authors have recognized this fact, but few authors actually provide better algorithms. We found that very few algorithms were actually able to improve the situation. Still, in 1985, Stewart [15] provided an algorithm which improves the robustness of Smith's algorithm, by considering all the ways of computing a product $a_1 \cdot a_2 \cdot a_3$ which appears in an intermediate expression of Smith's complex division. Stewart proved that if a_1, a_2 and a_3 are three floating point numbers such that the floating point representation of the product $a_1 \cdot a_2 \cdot a_3$ is a floating point number, then there is a way to compute the product which guarantees to avoid an overflow. There are exactly three ways of computing this product, namely $(a_1 \cdot a_2) \cdot a_3$, $a_1 \cdot (a_2 \cdot a_3)$ and $a_2 \cdot (a_1 \cdot a_3)$. Stewart's idea was to use the appropriate expression, by multiplying the number of largest magnitude with the one with smaller magnitude.

Stewart's algorithm works in more cases than Smith's. But the formulation of the algorithm is significantly more complicated. In the first edition of the algorithm [14], Stewart presented a wrong algorithm, which was presented in a fixed form one year later [15]. This increased complexity may have discouraged developers to use this improved algorithm.

In practice, Smith's method is still the most widely used algorithm. Moreover, we guess that most developers did not feel the need for improving their algorithm, probably by the lack of evidence that their complex division fails in situations of practical importance.

In this article, we present an improved Smith's algorithm, which performs significantly better than the original algorithm and is significantly simpler than Stewart's algorithm. More precisely, our algorithm does not require to compute the product $a_1 \cdot a_2 \cdot a_3$ in three different ways. In fact, we prove that one of the three ways is impossible, which simplifies the algorithm.

We compared our algorithm with other implementations and found that most algorithm which claimed for improved accuracy or improved performance, were in fact significantly less accurate than expected. More precisely, our numerical experiments suggest that the rate of failure of our improved algorithm is as low as Stewart's, and might be 4 orders of magnitude lower than a naive method, and 2 orders of magnitude lower than the other known implementations.

This paper is organized as follows. In the first section, we present the naive approach and present particular examples of failure of this method. We present Smith's 1962 method in the second section, while the third section focuses on known failure cases of Smith's method. In the fourth section, we present the lack of reproductibility of some particular complex divisions, caused by the use of extended precision on some processors. We present a review of other known complex division algorithms and show particular failure cases of these algorithms. Then we present our improved algorithm and a proof which states that the implementation is, in some sense, minimum. In the final section, we present numerical results and compare our improved implementation with 5 other algorithms.

2 The naive method

Assume that a, b, c and d are four real numbers. Consider the two complex numbers $a + ib$ and $c + id$, where i is the imaginary number which satisfies $i^2 = -1$. Assume that $c^2 + d^2$ is non zero. We are interested in the complex number $e + fi = \frac{a+ib}{c+id}$ where e and f are real numbers. The formula which allows to compute the real and imaginary parts of the division of these two complex numbers is

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}. \tag{1}$$

While the equation 1 is correct in exact arithmetic, it may fail when we consider floating point numbers. In Scilab, we use double precision floating point numbers. This corresponds to the binary 64 floating point numbers of the IEEE-2008 [5] standard, where the significand makes use of 52 binary digits and 11 binary digits in the exponent. Doubles have intrinsic limitations which can lead to rounding, overflow or underflow.

Hence, a naive implementation based on the previous formula can easily fail to produce an accurate result. The following Scilab function `naive` is a straightforward implementation of the previous formulas. It takes as input the complex numbers a and b, represented by their real and imaginary parts `a`, `b`, `c` and `d`. The function `naive` returns the complex number represented by its real and imaginary parts `e` and `f`.

```
function [e,f] = naive (a , b , c , d )
  den = c * c + d * d;
  e = (a * c + b * d) / den;
  f = (b * c - a * d) / den;
endfunction
```

Consider the complex division

$$\frac{1+i}{1+i10^{307}} \approx 1.0000000000000000 \cdot 10^{-307} - i1.0000000000000000 \cdot 10^{-307}, \tag{2}$$

which is accurate to the displayed digits. In fact, there are more that 300 zeros following the leading 1, so that the previous approximation is very accurate. The following Scilab session compares the naive implementation and Scilab's division operator.

```
-->  [e f] = naive ( 1.0 , 1.0 , 1.0 , 1.e307 )
 f  =
    0.
 e  =
    0.
-->  (1.0 + %i * 1.0)/(1.0 + %i * 1.e307)
 ans  =
    1.000-307 - 1.000-307i
```

In the previous case, the naive implementation does not produce any correct digit.

The last test involves small numbers in the denominator of the complex fraction. Consider the complex division

$$\frac{1+i}{10^{-307}+i10^{-307}} = \frac{1+i}{10^{-307}(1+i)} = 10^{307}. \tag{3}$$

In the following session, the first statement `ieee(2)` configures the IEEE system so that Inf and Nan numbers are generated instead of Scilab error messages.

```
-->ieee(2);
-->[e f] = naive ( 1.0 , 1.0 , 1.e-307 , 1.e-307 )
 f  =
   Nan
 e  =
   Inf
-->(1.0 + %i * 1.0)/(1.e-307 + %i * 1.e-307)
 ans  =
    1.000+307
```

We see that the naive implementation generates the IEEE numbers Nan and Inf, while the division operator produces the correct result.

3 Smith's method

In this section, we analyze Smith's method, which allows to produce an accurate division of two complex numbers.

In Scilab, the algorithm which allows to perform the complex division is done by the the *wwdiv* routine, which implements Smith's method [13]. Smith's algorithm is based on normalization, which allow to perform the complex division even if the input terms are large or small.

The starting point of the method is the mathematical definition 1, which is reproduced here for simplicity

$$\frac{a+ib}{c+id} = e + if = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}. \tag{4}$$

Smith's method is based on the rewriting of this formula in two different, but mathematically equivalent, formulas. We have seen that the term $c^2 + d^2$ may generate overflows or underflows. This is caused by intermediate expressions which magnitudes are larger than necessary. The previous numerical experiments suggest that, provided that we had simplified the calculation, the intermediate expressions would not have been unnecessary large. Hence, Smith's idea consists in rewriting the expressions so that we avoid writing the term $c^2 + d^2$.

The following `smith` function implements Smith's method in the Scilab language.

```
function [e,f] = smith ( a , b , c , d )
  if ( abs(d) <= abs(c) ) then
    r = d/c;
    den = c + d * r;
    e = (a + b * r) / den;
    f = (b - a * r) / den;
  else
    r = c/d;
    den = c * r + d;
    e = (a * r + b) / den;
    f = (b * r - a) / den;
  end
endfunction
```

It is easy to check that Smith's method performs very well for the relatively difficult complex divisions that we met earlier. In fact, Smith's method is not as robust as one might think and this is the topic of the next section.

4 Robustness of Smith's algorithm

In this section, we present an example where Smith's method does not perform as expected.

The following example is inspired by an example by Stewart's in [14]. While Stewart gives an example based on a machine with an exponent range ± 99, we consider an example which is based on Scilab's doubles. Consider the complex division

$$\frac{10^{307}+i10^{-307}}{10^{204}+i10^{-204}} \approx 1.000000000000000 \cdot 10^{103} - i1.000000000000000 \cdot 10^{-305}, \tag{5}$$

which is accurate to the displayed digits. In fact, there are more that 100 zeros following the leading 1, so that the previous approximation is very accurate. The following Scilab session compares the naive implementation, Smith's method and Scilab's division operator. The session is performed with Scilab v5.2.0 under a 32 bits Windows using a Intel Xeon processor.

```
-->[e f] = naive ( 1.e307 , 1.e-307 , 1.e204 , 1.e-204 )
 f  =
    0.
 e  =
   Nan
-->[e f] = smith ( 1.e307 , 1.e-307 , 1.e204 , 1.e-204 )
 f  =
    0.
 e  =
    1.000+103
-->(1.e307 + %i * 1.e-307)/(1.e204 + %i * 1.e-204)
 ans  =
    1.000+103 - 1.000-305i
```

In the previous case, the naive implementation does not produce any correct digit, as expected. Smith's method, produces a correct real part, but an inaccurate imaginary part. Once again, Scilab's division operator provides the correct answer.

We check that Smith's formula is not accurate in this case. Indeed, it performs the following steps.

```
if ( abs(d) = 1.e-204 <= abs(c) = 1.e204 ) > test true
  r = d/c = 1.e-204 / 1.e204 = 0
  den  = c + d * r = 1.e204 +  1.e-204 * 0 = 1.e204
  e = (a + b * r) / den = (1.e307 + 1.e-307 * 0) / 1e204
    = 1.e307 / 1.e204 = 1.e103
  f = (b - a * r) / den = (1.e-307 - 1.e307 * 0) / 1e204
    = 1.e-307 / 1.e204 = 0
```

We see that the variable `r` underflows, so that it is represented by zero. This simplifies the denominator `den`, but this variable is still correctly computed, because it is dominated the term `c`. The real part `e` is still accurate, because, once again, the computation is dominated by the

Scilab v5.2.0 release	Windows 32 bits	1.000+103 - 1.000-305i
Scilab v5.2.0 release	Windows 64 bits	1.000+103
Scilab v5.2.0 debug	Windows 32 bits	1.000+103
Scilab v5.1.1 release	Windows 32 bits	1.000+103
Scilab v4.1.2 release	Windows 32 bits	1.000+103
Scilab v5.2.0 release	Linux 32 bits	1.000+103 - 1.000-305i
Scilab v5.1.1 release	Linux 32 bits	1.000+103 - 1.000-305i
Octave v3.0.3	Windows 32 bits	1.0000e+103
Matlab 2008	Windows 32 bits	1.0000e+103 -1.0000e-305i
Matlab 2008	Windows 64 bits	1.0000e+103
FreeMat v3.6	Windows 32 bits	1.0000e+103 -1.0000e-305i

Figure 1: Result of the complex division `(1.e307 + %i * 1.e-307)/(1.e204 + %i * 1.e-204)` on various softwares and operating systems.

term `a`. The imaginary part `f` is wrong, because this term should be dominated by the term `a*r`. Since `r` underflows, it is represented by zero, which completely changes the result of the expression `b-a*r`, which is now equal to `b`. Therefore, the result is equal to `1.e-307 / 1.e204`, which underflows to zero.

Since Scilab makes use of Smith's formula, it should fail in this case. But we have seen that Scilab is still able to perform accurately. This unexpected accuracy is analyzed in the next section.

4.1 The x87 issue

We now analyze why Scilab's division operator performs accurately in this case. Indeed, the formula used by Scilab is based on Smith's method and we proved that this method fails in this case, when we use double floating point numbers. Therefore, we experienced here an unexpected high accuracy.

We performed this particular complex division over several common computing systems such as various versions of Scilab, Octave, Matlab and FreeMat on various operating systems. The results are presented in figure 1. Notice that, on Matlab, Octave and FreeMat, the syntax is different and we used the expression `(1.e307 + i * 1.e-307)/(1.e204 + i * 1.e-204)`.

The reason of the discrepancies of the results is the following [11, 10]. The processor being used may offer an internal precision that is wider than the precision of the variables of a program. Indeed, processors of the IA32 architecture (Intel 386, 486, Pentium etc. and compatibles) feature a floating-point unit often known as "x87". This unit has 80-bit registers in "double extended" format with a 64-bit mantissa and a 15-bit exponent. The most usual way of generating code for the IA32 is to hold temporaries - and, in optimized code, program variables - in the x87 registers. Hence, the final result of the computations depend on how the compiler allocates registers. Since the double extended format of the x87 unit uses 15 bits for the exponent, it can store floating point numbers associated with binary exponents from $2^{-16382} \approx 10^{-4932}$ up to $2^{16383} \approx 10^{4931}$, which is much larger than the exponents from the 64-bits double precision floating point numbers (ranging from $2^{-1022} \approx 10^{-308}$ up to $2^{1023} \approx 10^{307}$). Therefore, the computations performed with the x87 unit are less likely to generate underflows and overflows. On the other hand, SSE2 extensions introduced one 128-bit packed floating-point data type. This 128-bit data type consists of two

IEEE 64-bit double-precision floating-point values packed into a double quadword.

Depending on the compilers options used to generate the binary, the result may use either the x87 unit (with 80-bits registers) or the SSE unit. Under Windows 32 bits, Scilab v5.2.0 is compiled with the "/arch:IA32" option [2], which allows Scilab to run on older Pentium computers that does not support SSE2. In this situation, Scilab may use the x87 unit. Under Windows 64 bits, Scilab uses the SSE2 unit so that the result is based on double precision floating point numbers only. Under Linux, Scilab is compiled with gcc [3], where the behavior is driven by the -mfpmath option. The default value of this option for i386 machines is to use the 387 floating point co-processor while, for x86_64 machines, the default is to use the SSE instruction set.

5 A review of algorithms

The 1962 paper by R. Smith [13] describes the algorithm which is used in Scilab. An analysis of Hough, cited by Coonen [1] and Stewart [14] shows that when the algorithm works, it returns a computed value $\overline{z}$ satisfying

$$|\overline{z} - z| \leq \epsilon |z|, \tag{6}$$

where z is the exact complex division result and ϵ is of the same order of magnitude as the rounding unit for the arithmetic in question.

The limits of Smith's method have been analyzed by Stewart's in [14]. The paper separates the relative error of the complex numbers and the relative error made on real and imaginary parts. Stewart's algorithm is based on a theorem which states that if $x_1 \dots x_n$ are n floating point representable numbers, and if their product is also a representable floating point number, then the product $\min_{i=1,n}(x_i).\max_{i=1,n}(x_i)$ is also representable. The algorithm uses that theorem to perform a correct computation.

TODO : Stewart's algorithm

TODO : a failure of Stewart's

In [9], Li et al. present an improved complex division algorithm with scaling. The section 6.1, "Environmental Enquiries", presents Smith's algorithm. The authors state that this algorithm can suffer from intermediate underflow. As complex division occurs rarely in the BLAS, the authors have chosen to have a more careful implementation. This implementation scales the numerator and denominator if they are two small or too large. An error bound is presented for this algorithm, which is presented in the appendix B, "Smith's Complex Division Algorithm with Scaling" of the technical report [8] (but not in the paper [9]).

TODO : the Li et al. algorithm

TODO : a failure case of Li et al.

In the ISO/IEC 9899:TC3 C Committee Draft [6], the section G.5.1 "Multiplicative operators", the authors present a `_Cdivd` function which implements the complex division. Their implementation only scales the denominator c^2+d^2. This scaling is based on a power of 2, which avoid extra rounding. Only in the case of an IEEE exceptions, the algorithm recompute the division, taking into account for Nans and Infinities. According to the authors, this solves the main overflow and underflow problem. The code does not defend against overflow and underflow in the calculation of the numerator. According to Kahan [7] (in the Appendix "Over/Underflow Undermines Complex Number Division in Java"), this code is due to Jim Thomas and Fred Tydeman.

TODO : the C algorithm

TODO : a failure case of the C algorithm

In the exercise 25.2 of the chapter 25 "Software issues in floating point arithmetic" of [4], Nicolas Higham makes a link between the complex division and the Gaussian elimination. He suggest that Smith's algorithm can be derived from appliying the Gaussian elimination algorithm with partial pivoting obtained from $(c+id)(e+if) = a+ib$.

Priest published in 2004 [12] a complex division algorithm based on scaling. This scaling is designed to avoid overflow and harmful underflow. The scaling requires only four floating point multiplications and a small amount of integer arithmetic to compute the scale factor. Priest shows that choosing a scaling factor close to $|w|^{-3/4}$ works well in most situations.

TODO : Priest's algorithm

TODO : a failure case of Priest's algorithm

6 The algorithm

In this section, we present an improved complex division algorithm and prooves that the algorithm is, in some sense, minimum.

The following Scilab function is an improved complex division.

```
function r = compdiv_improved ( x, y )
    a = real(x)
    b = imag(x)
    c = real(y)
    d = imag(y)
    if ( abs(d) <= abs(c) ) then
        r = d/c
        t = 1/(c + d * r)
        if (r == 0) then
            e = (a + d * (b/c)) * t
            f = (b - d * (a/c)) * t
        else
            e = (a + b * r) * t
            f = (b - a * r) * t
        end
    else
        r = c/d
        t = 1/(c * r + d  )
        if (r == 0) then
            e = (c * (a/d) + b) * t
            f = (c * (b/d) - a) * t
        else
            e = (a * r + b) * t
            f = (b * r - a) * t
        end
    end
    r = complex(e,f)
endfunction
```

Assume that $|d| \leq |c|$. In this case, Smith's method was to compute

$$e = (a + b * r) * t; f = (b - a * r) * t; \tag{7}$$

We have seen that Smith's formulat may fail when the expression $r = d/c$ underflows, so that f is finally computed as zero, instead of being non-zero.

The expression for f is $f = (b - a * r) * t$ and we have seen that it may fail in the case where r underflows to zero, so that the product $a * r$ is evaluated as zero. One possible approach is to consider different ways of evaluating the expression $a * r = a * (d/c)$. There are three different ways of evaluating the product $a * d/c$, namely $a * r = a * (d/c)$, $d * (a/c)$ and $(d * a)/c$.

In the case where r underflows, it may happen that one of the two expressions $d * (a/c)$ and $(d * a)/c$ can lead to an accurate result. All in all, this leads to three possible ways of evaluating e and f from the equations:

$$e = (a + d * (b/c)) * t, \qquad f = (b - d * (a/c)) * t \tag{8}$$

$$e = (a + b * r) * t, \qquad f = (b - a * r) * t \tag{9}$$

$$e = (a + (d * b)/c) * t, \qquad f = (b - (d * a)/c)) * t. \tag{10}$$

7 Numerical experiments

8 Conclusion

References

[1] J.T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981.

[2] Martyn J. Corden and David Kreitzer. Consistency of floating-point results using the intel compiler or why doesn't my application always give the same answer? Technical report, Intel Corporation, Software Solutions Group, 2009.

[3] Free Software Foundation. The gnu compiler collection. Technical report, 2008.

[4] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[5] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[6] ISO/IEC. Programming languages - c, iso/iec 9899:tc3. Technical report, 2007. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[7] William Kahan. Marketing versus mathematics. www.cs.berkeley.edu/~wkahan/MktgMath.ps, 2000.

[8] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. C. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision blas. www.netlib.org/lapack/lawnspdf/lawn149.pdf, 2000.

[9] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[10] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[11] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[12] Douglas M. Priest. Efficient scaling for complex division. *ACM Trans. Math. Softw.*, 30(4):389–401, 2004.

[13] Robert L. Smith. Algorithm 116: Complex division. *Commun. ACM*, 5(8):435, 1962.

[14] G. W. Stewart. A note on complex division. *ACM Trans. Math. Softw.*, 11(3):238–241, 1985.

[15] G. W. Stewart. Corrigendum: "A note on complex division". 12(3):285–285, September 1986.