Scilab is not naïve

Michael Baudin

February 2009

Abstract

Most of the time, the mathematical formula is directly used in the Scilab source code. But, in many algorithms, some additionnal work is performed, which takes into account the fact that the computer do not process mathematical real values, but performs computations with their floating point representation. The goal of this article is to show that, in many situations, Scilab is not naïve and use algorithms which have been specifically tailored for floating point computers. We analyse in this article the particular case of the quadratic equation, the complex division and the numerical derivatives, and show that one these examples, the naïve algorithm is not sufficiently accurate.

Contents

1 Introduction

There is a huge space between the numerical formula, as presented in basic mathematical books, and the implementation of the numerical algorithm, and especially in Scilab. While the mathematic theory deals with the correctness of the formulas, *Scilab take cares with your numbers.*

This difficulty is generated by the fact that, while the mathematics treat with *real* numbers, the computer deals with their *floating point representations.*

A central reference on this subject is the article by Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", [7]. If one focuses on numerical algorithms, the "Numerical Recipes" [14], is another good sources of solutions for that problem. The work of Kahan is also central in this domain, for example [11].

In this article, we will show examples of these problems by using the following theoric and experimental approach.

1. First, we will derive the basic theory at the core of a numerical formula.

2. Then we will implement it in Scilab and compare with the result given by the primitive provided by Scilab. As we will see, some particular cases do not work well with our formula, while the Scilab primitive computes a correct result.

3. Then we will analyse the *reasons* of the differences.

When we compute errors, we use the relative error formula

$$e_r = \frac{|x_c - x_e|}{|x_e|}, \qquad x_e \neq 0 \tag{1}$$

where $x_c \in \mathbb{R}$ is the computed value, and $x_e \in \mathbb{R}$ is the expected value, i.e. the mathematically exact result. The relative error is linked with the number of significant digits in the computed value x_c. For example, if the relative error $e_r = 10^{-6}$, then the number of significant digits is 6.

When the expected value is zero, the relative error cannot be computed, and we then use the absolute error

$$e_a = |x_c - x_e|. \tag{2}$$

Before getting into the details, it is important to know that real variables in the Scilab language are stored in *double precision* variables. Since Scilab is following the IEEE 754 standard, that means that real variables are stored with 64 bits precision. As we shall see later, this has a strong influence on the results.

2 Quadratic equation

In this section, we detail the computation of the roots of a quadratic polynomial. As we shall see, there is a whole world from the mathematics formulas to the implementation of such computations. In the first part, we briefly report the formulas which allow to compute the real roots of a quadratic equation with real coefficients. We then present the naïve algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naïve algorithm with the *roots* Scilab primitive. In the third part, we analyse why and how floating point numbers must be taken into account when the implementation of such roots is required.

2.1 Theory

We consider the following quadratic equation, with real coefficients $a, b, c \in \mathbb{R}$ [2, 1, 3] :

$$ax^2 + bx + c = 0. \tag{3}$$

The real roots of the quadratic equations are

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \tag{4}$$

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{5}$$

with the hypothesis that the discriminant $\Delta = b^2 - 4ac$ is positive.

The naive, simplified, algorithm to use to compute the roots of the quadratic is presented in figure 1.

$\Delta \leftarrow b^2 - 4ac$
$s \leftarrow \sqrt{\Delta}$
$x_- \leftarrow (-b - s)/(2a)$
$x_+ \leftarrow (-b + s)/(2a)$

Figure 1: Naive algorithm to compute the real roots of a quadratic equation

2.2 Experiments

The following Scilab function is a straitforward implementation of the previous formulas.

```
1 function r=myroots(p)
2   c=coeff(p,0);
3   b=coeff(p,1);
4   a=coeff(p,2);
5   r=zeros(2,1);
6   r(1)=(-b+sqrt(b^2-4*a*c))/(2*a);
7   r(2)=(-b-sqrt(b^2-4*a*c))/(2*a);
8 endfunction
```

The goal of this section is to show that some additionnal work is necessary to compute the roots of the quadratic equation with sufficient accuracy. We will especially pay attention to rounding errors and overflow problems. In this section, we show that the *roots* command of the Scilab language is not *naive*, in the sense that it takes into account for the floating point implementation details that we will see in the next section.

2.2.1 Rounding errors

We analyse the rounding errors which are appearing when the discriminant of the quadratic equation is such that $b^2 \approx 4ac$. We consider the following quadratic equation

$$\epsilon x^2 + (1/\epsilon)x - \epsilon = 0 \tag{6}$$

with $\epsilon = 0.0001 = 10^{-4}$.

The two real solutions of the quadratic equation are

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \approx -1/\epsilon^2, \tag{7}$$

$$x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \approx \epsilon^2 \tag{8}$$

The following Scilab script shows an example of the computation of the roots of such a polynomial with the *roots* primitive and with a naive implementation. Only the positive root $x_+ \approx \epsilon^2$ is considered in this test (the x_- root is so that $x_- \to -\infty$ in both implementations).

```
1  p=poly([-0.0001 10000.0 0.0001],"x","coeff");
2  e1 = 1e-8;
3  roots1 = myroots(p);
4  r1 = roots1(1);
5  roots2 = roots(p);
6  r2 = roots2(1);
7  error1 = abs(r1-e1)/e1;
8  error2 = abs(r2-e1)/e1;
9  printf("Expected : %e\n", e1);
10 printf("Naive method : %e (error=%e)\n", r1,error1);
11 printf("Scilab method : %e (error=%e)\n", r2, error2);
```

The script then prints out :

```
Expected : 1.000000e-008
Naive method : 9.094947e-009 (error=9.050530e-002)
Scilab method : 1.000000e-008 (error=1.654361e-016)
```

The result is astonishing, since the naive root has no correct digit and a relative error which is 14 orders of magnitude greater than the relative error of the Scilab root.

The explanation for such a behaviour is that the expression of the positive root is the following

$$x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon} \tag{9}$$

and is numerically evalutated as

```
\sqrt{1/\epsilon^2+4\epsilon^2} = 10000.000000000001818989
```

As we see, the first digits are correct, but the last digits are polluted with rounding errors. When the expression $-1/\epsilon+\sqrt{1/\epsilon^2+4\epsilon^2}$ is evaluated, the following computations are performed :

```
-1/\epsilon+ \sqrt{1/\epsilon^2+4\epsilon^2}
  = -10000.0 + 10000.000000000001818989
  = 0.000000000018189894035
```

The user may think that the result is extreme, but it is not. Reducing furter the value of ϵ down to $\epsilon = 10^{-11}$, we get the following output :

```
Expected : 1.000000e-022
Naive method : 0.000000e+000 (error=1.000000e+000)
Scilab method : 1.000000e-022 (error=1.175494e-016)
```

The relative error is this time 16 orders of magnitude greater than the relative error of the Scilab root. In fact, the naive implementation computes a false root x_+ even for a value of epsilon equal to $\epsilon = 10^-3$, where the relative error is 7 times greater than the relative error produced by the *roots* primitive.

2.2.2 Overflow

In this section, we analyse the overflow exception which is appearing when the discriminant of the quadratic equation is such that $b^2 >> 4ac$. We consider the following quadratic equation

$$x^2 + (1/\epsilon)x + 1 = 0 \tag{10}$$

with $\epsilon \to 0$.

The roots of this equation are

$$x_- \approx -1/\epsilon \to -\infty, \qquad \epsilon \to 0 \tag{11}$$

$$x_+ \approx -\epsilon \to 0^-, \qquad \epsilon \to 0 \tag{12}$$

To create a difficult case, we search ϵ so that $1/\epsilon^2 = 10^{310}$, because we know that 10^{308} is the maximum value available with double precision floating point numbers. The solution is $\epsilon = 10^{-155}$.

The following Scilab script shows an example of the computation of the roots of such a polynomial with the *roots* primitive and with a naive implementation.

```
1  // Test #3 : overflow because of b
2  e=1.e-155
3  a = 1;
4  b = 1/e;
5  c = 1;
6  p=poly([c b a],"x","coeff");
7  expected = [-e;-1/e];
8  roots1 = myroots(p);
9  roots2 = roots(p);
10 error1 = abs(roots1-expected)/norm(expected);
11 error2 = abs(roots2-expected)/norm(expected);
12 printf("Expected : %e %e\n", expected(1),expected(2));
13 printf("Naive method : %e %e (error=%e)\n", roots1(1),roots1(2),error1);
14 printf("Scilab method : %e %e (error=%e)\n", roots2(1),roots2(2), error2);
```

The script then prints out :

```
Expected : -1.000000e-155 -1.000000e+155
Naive method : Inf Inf (error=Nan)
Scilab method : -1.000000e-155 -1.000000e+155 (error=0.000000e+000)
```

As we see, the $b^2 - 4ac$ term has been evaluated as $1/\epsilon^2 - 4$, which is approximately equal to 10^{310}. This number cannot be represented in a floating point number. It therefore produces the IEEE overflow exception and set the result as *Inf*.

2.3 Explanations

The technical report by G. Forsythe [5] is especially interesting on that subject. The paper by Goldberg [7] is also a good reference for the quadratic equation. One can also consult the experiments performed by Nievergelt in [13].

The following tricks are extracted from the *quad* routine of the *RPOLY* algorithm by Jenkins [9]. This algorithm is used by Scilab in the roots primitive, where a special case is handled when the degree of the equation is equal to 2, i.e. a quadratic equation.

2.3.1 Properties of the roots

One can easily show that the sum and the product of the roots allow to recover the coefficients of the equation which was solve. One can show that

$$x_- + x_+ = \frac{-b}{a} \tag{13}$$

$$x_- x_+ = \frac{c}{a} \tag{14}$$

Put in another form, one can state that the computed roots are solution of the normalized equation

$$x^2 - \left(\frac{x_- + x_+}{a}\right) x + x_- x_+ = 0 \tag{15}$$

Other transformation leads to an alternative form for the roots. The original quadratic equation can be written as a quadratic equation on $1/x$

$$c(1/x)^2 + b(1/x) + a = 0 \tag{16}$$

Using the previous expressions for the solution of $ax^2+bx+c = 0$ leads to the following expression of the roots of the quadratic equation when the discriminant is positive

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{17}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{18}$$

These roots can also be computed from 4, with the multiplication by $-b + \sqrt{b^2 - 4ac}$.

2.3.2 Conditionning of the problem

The conditionning of the problem may be evaluated with the computation of the partial derivatives of the roots of the equations with respect to the coefficients. These partial derivatives measure the sensitivity of the roots of the equation with respect to small errors which might pollute the coefficients of the quadratic equations.

In the following, we note $x_- = \frac{-b-\sqrt{\Delta}}{2a}$ and $x_+ = \frac{-b+\sqrt{\Delta}}{2a}$ when $a \neq 0$. If the discriminant is stricly positive and $a \neq 0$, i.e. if the roots of the quadratic are real, the partial derivatives of the roots are the following :

$$\frac{\partial x_-}{\partial a} = \frac{c}{a\sqrt{\Delta}} + \frac{b+\sqrt{\Delta}}{2a^2}, \qquad a \neq 0, \qquad \Delta \neq 0 \tag{19}$$

$$\frac{\partial x_+}{\partial a} = -\frac{c}{a\sqrt{\Delta}} + \frac{b-\sqrt{\Delta}}{2a^2} \tag{20}$$

$$\frac{\partial x_-}{\partial b} = \frac{-1-b/\sqrt{\Delta}}{2a} \tag{21}$$

$$\frac{\partial x_+}{\partial b} = \frac{-1+b/\sqrt{\Delta}}{2a} \tag{22}$$

$$\frac{\partial x_-}{\partial c} = \frac{1}{\sqrt{\Delta}} \tag{23}$$

$$\frac{\partial x_+}{\partial c} = -\frac{1}{\sqrt{\Delta}} \tag{24}$$

If the discriminant is zero, the partial derivatives of the double real root are the following :

$$\frac{\partial x_\pm}{\partial a} = \frac{b}{2a^2}, \qquad a \neq 0 \tag{25}$$

$$\frac{\partial x_\pm}{\partial b} = \frac{-1}{2a} \tag{26}$$

$$\frac{\partial x_\pm}{\partial c} = 0 \tag{27}$$

The partial derivates indicate that if $a \approx 0$ or $\Delta \approx 0$, the problem is ill-conditionned.

2.3.3 Floating-Point implementation : fixing rounding error

In this section, we show how to compute the roots of a quadratic equation with protection against rounding errors, protection against overflow and a minimum amount of multiplications and divisions.

Few but important references deals with floating point implementations of the roots of a quadratic polynomial. These references include the important paper [7] by Golberg, the Numerical Recipes [14], chapter 5, section 5.6 and [6], [13], [11].

The starting point is the mathematical solution of the quadratic equation, depending on the sign of the discriminant $\Delta = b^2 - 4ac$:

- If $\Delta > 0$, there are two real roots,

$$x_\pm = \frac{-b \pm \sqrt{\Delta}}{2a}, \qquad a \neq 0 \tag{28}$$

- If $\Delta = 0$, there are one double root,

$$x_\pm = -\frac{b}{2a}, \qquad a \neq 0 \tag{29}$$

- If $\Delta < 0$,

$$x_\pm = \frac{-b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a}, \qquad a \neq 0 \tag{30}$$

In the following, we make the hypothesis that $a \neq 0$.

The previous experiments suggest that the floating point implementation must deal with two different problems :

- rounding errors when $b^2 \approx 4ac$ because of the cancelation of the terms which have opposite signs,
- overflow in the computation of the discriminant Δ when b is large in magnitude with respect to a and c.

When $\Delta > 0$, the rounding error problem can be splitted in two cases

- if $b < 0$, then $-b + \sqrt{b^2 - 4ac}$ may suffer of rounding errors,
- if $b > 0$, then $-b - \sqrt{b^2 - 4ac}$ may suffer of rounding errors.

Obviously, the rounding problem will not appear when $\Delta < 0$, since the complex roots do not use the sum $-b + \sqrt{b^2 - 4ac}$. When $\Delta = 0$, the double root does not cause further trouble. The rounding error problem must be solved only when $\Delta > 0$ and the equation has two real roots.

A possible solution may found in combining the following expressions for the roots

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \tag{31}$$

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{32}$$

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \tag{33}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \tag{34}$$

The trick is to pick the formula so that the sign of b is the same as the sign of the square root. The following choice allow to solve the rounding error problem

- compute x_- : if $b < 0$, then compute x_- from 32, else (if $b > 0$), compute x_- from 31,
- compute x_+ : if $b < 0$, then compute x_+ from 33, else (if $b > 0$), compute x_+ from 34.

The solution of the rounding error problem can be adressed, by considering the modified Fagnano formulas

$$x_1 = -\frac{2c}{b + sgn(b)\sqrt{b^2 - 4ac}}, \tag{35}$$

$$x_2 = -\frac{b + sgn(b)\sqrt{b^2 - 4ac}}{2a}, \tag{36}$$

where

$$sgn(b) = \begin{cases} 1, \text{ if } b \geq 0, \\ -1, \text{ if } b < 0, \end{cases} \tag{37}$$

The roots $x_{1,2}$ correspond to $x_{+,-}$ so that if $b < 0$, $x_1 = x_-$ and if $b > 0$, $x_1 = x_+$. On the other hand, if $b < 0$, $x_2 = x_+$ and if $b > 0$, $x_2 = x_-$.

An additionnal remark is that the division by two (and the multiplication by 2) is exact with floating point numbers so these operations cannot be a source of problem. But it is interesting to use $b/2$, which involves only one division, instead of the three multiplications $2 * c$, $2 * a$ and $4 * a * c$. This leads to the following expressions of the real roots

$$x_- = -\frac{c}{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}, \tag{38}$$

$$x_+ = -\frac{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}{a}, \tag{39}$$

which can be simplified into

$$b' = b/2 \tag{40}$$

$$h = -\left(b' + sgn(b)\sqrt{b'^2 - ac}\right) \tag{41}$$

$$x_1 = \frac{c}{h}, \tag{42}$$

$$x_2 = \frac{h}{a}, \tag{43}$$

where the discriminant is positive, i.e. $b'^2 - ac > 0$.

One can use the same value $b' = b/2$ with the complex roots in the case where the discriminant is negative, i.e. $b'^2 - ac < 0$:

$$x_1 = -\frac{b'}{a} - i\frac{\sqrt{ac - b'^2}}{a}, \tag{44}$$

$$x_2 = -\frac{b'}{a} + i\frac{\sqrt{ac - b'^2}}{a}, \tag{45}$$

A more robust algorithm, based on the previous analysis is presented in figure 2. By comparing 1 and 2, we can see that the algorithms are different in many points.

2.3.4 Floating-Point implementation : fixing overflow problems

The remaining problem is to compute $b'^2 - ac$ without creating unnecessary overflows.

if $a = 0$ **then**
  if $b = 0$ **then**
    $x_- \leftarrow 0$
    $x_+ \leftarrow 0$
  else
    $x_- \leftarrow -c/b$
    $x_+ \leftarrow 0$
  end if
else if $c = 0$ **then**
  $x_- \leftarrow -b/a$
  $x_+ \leftarrow 0$
else
  $b' \leftarrow b/2$
  $\Delta \leftarrow b'^2 - ac$
  if $\Delta < 0$ **then**
    $s \leftarrow \sqrt{-\Delta}$
    $x_1^R \leftarrow -b'/a$
    $x_1^I \leftarrow -s/a$
    $x_2^R \leftarrow x_-^R$
    $x_2^I \leftarrow -x_1^I$
  else if $\Delta = 0$ **then**
    $x_1 \leftarrow -b'/a$
    $x_2 \leftarrow x_2$
  else
    $s \leftarrow \sqrt{\Delta}$
    if $b > 0$ **then**
      $g = 1$
    else
      $g = -1$
    end if
    $h = -(b' + g * s)$
    $x_1 \leftarrow c/h$
    $x_2 \leftarrow h/a$
  end if
end if

Figure 2: A more robust algorithm to compute the roots of a quadratic equation

Notice that a small improvment has allread been done : if $|b|$ is close to the upper bound 10^{154}, then $|b'|$ may be less difficult to process since $|b'| = |b|/2 < |b|$. One can then compute the square root by using normalization methods, so that the overflow problem can be drastically reduced. The method is based on the fact that the term $b'^2 - ac$ can be evaluted with two equivalent formulas

$$b'^2 - ac = b'^2 [1 - (a/b')(c/b')] \quad (46)$$
$$b'^2 - ac = c [b'(b'/c) - a] \quad (47)$$

- If $|b'| > |c| > 0$, then the expression involving $(1 - (a/b')(c/b'))$ is so that no overflow is possible since $|c/b'| < 1$ and the problem occurs only when b is large in magnitude with respect to a and c.
- If $|c| > |b'| > 0$, then the expression involving $(b'(b'/c) - a)$ should limit the possible overflows since $|b'/c| < 1$.

These normalization tricks are similar to the one used by Smith in the algorithm for the division of complex numbers [17].

3 Numerical derivatives

In this section, we detail the computation of the numerical derivative of a given function.

In the first part, we briefly report the first order forward formula, which is based on the Taylor theorem. We then present the naïve algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naïve algorithm with the *derivative* Scilab primitive. In the third part, we analyse why and how floating point numbers must be taken into account when the numerical derivatives are to compute.

A reference for numerical derivates is [4], chapter 25. "Numerical Interpolation, Differentiation and Integration" (p. 875). The webpage [16] and the book [14] give results about the rounding errors.

3.1 Theory

The basic result is the Taylor formula with one variable [8]

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f''''(x) + \mathcal{O}(h^5) \quad (48)$$

If we write the Taylor formulae of a one variable function $f(x)$

$$f(x+h) \approx f(x) + h\frac{\partial f}{\partial x} + \frac{h^2}{2}f''(x) \quad (49)$$

we get the forward difference which approximates the first derivate at order 1

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + \frac{h}{2}f''(x) \quad (50)$$

The naive algorithm to compute the numerical derivate of a function of one variable is presented in figure 3.

$f'(x) \leftarrow (f(x+h) - f(x))/h$

Figure 3: Naive algorithm to compute the numerical derivative of a function of one variable

3.2 Experiments

The following Scilab function is a straitforward implementation of the previous algorithm.

```
1 function fp = myfprime(f,x,h)
2   fp = (f(x+h) - f(x))/h;
3 endfunction
```

In our experiments, we will compute the derivatives of the square function $f(x) = x^2$, which is $f'(x) = 2x$. The following Scilab script implements the square function.

```
1 function y = myfunction (x)
2   y = x*x;
3 endfunction
```

The most naïve idea is that the computed relative error is small when the step h is small. Because *small* is not a priori clear, we take $\epsilon \approx 10^{-16}$ in double precision as a good candidate for *small*. In the following script, we compare the computed relative error produced by our naïve method with step $h = \epsilon$ and the *derivative* primitive with default step.

```
x = 1.0;
fpref = derivative(myfunction,x,order=1);
e = abs(fpref-2.0)/2.0;
mprintf("Scilab f''=%e, error=%e\n", fpref,e);
h = 1.e-16;
fp = myfprime(myfunction,x,h);
e = abs(fp-2.0)/2.0;
mprintf("Naive f''=%e, h=%e, error=%e\n", fp,h,e);
```

When executed, the previous script prints out :

```
Scilab f'=2.000000e+000, error=7.450581e-009
Naive f'=0.000000e+000, h=1.000000e-016, error=1.000000e+000
```

Our naïve method seems to be quite inaccurate and has not even 1 significant digit ! The Scilab primitive, instead, has 9 significant digits.

Since our faith is based on the truth of the mathematical theory, some deeper experiments must be performed. We then make the following experiment, by taking an initial step $h = 1.0$ and then dividing h by 10 at each step of a loop with 20 iterations.

```
x = 1.0;
fpref = derivative(myfunction,x,order=1);
e = abs(fpref-2.0)/2.0;
mprintf("Scilab f''=%e, error=%e\n", fpref,e);
h = 1.0;
for i=1:20
  h=h/10.0;
  fp = myfprime(myfunction,x,h);
  e = abs(fp-2.0)/2.0;
  mprintf("Naive f''=%e, h=%e, error=%e\n", fp,h,e);
end
```

Scilab then produces the following output.

```
Scilab f'=2.000000e+000, error=7.450581e-009
Naive f'=2.100000e+000, h=1.000000e-001, error=5.000000e-002
Naive f'=2.010000e+000, h=1.000000e-002, error=5.000000e-003
Naive f'=2.001000e+000, h=1.000000e-003, error=5.000000e-004
Naive f'=2.000100e+000, h=1.000000e-004, error=5.000000e-005
Naive f'=2.000010e+000, h=1.000000e-005, error=5.000007e-006
Naive f'=2.000001e+000, h=1.000000e-006, error=4.999622e-007
Naive f'=2.000000e+000, h=1.000000e-007, error=5.054390e-008
Naive f'=2.000000e+000, h=1.000000e-008, error=6.077471e-009
Naive f'=2.000000e+000, h=1.000000e-009, error=8.274037e-008
Naive f'=2.000000e+000, h=1.000000e-010, error=8.274037e-008
Naive f'=2.000000e+000, h=1.000000e-011, error=8.274037e-008
Naive f'=2.000178e+000, h=1.000000e-012, error=8.890058e-005
Naive f'=1.998401e+000, h=1.000000e-013, error=7.992778e-004
```

```
Naive f'=1.998401e+000, h=1.000000e-014, error=7.992778e-004
Naive f'=2.220446e+000, h=1.000000e-015, error=1.102230e-001
Naive f'=0.000000e+000, h=1.000000e-016, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-017, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-018, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-019, error=1.000000e+000
Naive f'=0.000000e+000, h=1.000000e-020, error=1.000000e+000
```

We see that the relative error begins by decreasing, and then is increasing. Obviously, the optimum step is approximately $h = 10^{-8}$, where the relative error is approximately $e_r = 6.10^{-9}$. We should not be surprised to see that Scilab has computed a derivative which is near the optimum.

3.3 Explanations

3.3.1 Floating point implementation

With a floating point computer, the total error that we get from the forward difference approximation is (skipping the multiplication constants) the sum of the linearization error $E_l = h$ (i.e. the $\mathcal{O}(h)$ term) and the rounding error $rf(x)$ on the difference $f(x+h) - f(x)$

$$E = \frac{rf(x)}{h} + \frac{h}{2}f''(x) \tag{51}$$

When $h \to \infty$, the error is then the sum of a term which converges toward $+\infty$ and a term which converges toward 0. The total error is minimized when both terms are equal. With a single precision computation, the rounding error is $r = 10^{-7}$ and with a double precision computation, the rounding error is $r = 10^{-16}$. We make here the assumption that the values $f(x)$ and $f''(x)$ are near 1 so that the error can be written

$$E = \frac{r}{h} + h \tag{52}$$

We want to compute the step h from the rounding error r with a step satisfying

$$h = r^{\alpha} \tag{53}$$

for some $\alpha > 0$. The total error is therefore

$$E = r^{1-\alpha} + r^{\alpha} \tag{54}$$

The total error is minimized when both terms are equal, that is, when the exponents are equal $1 - \alpha = \alpha$ which leads to

$$\alpha = \frac{1}{2} \tag{55}$$

We conclude that the step which minimizes the error is

$$h = r^{1/2} \tag{56}$$

and the associated error is

$$E = 2r^{1/2} \tag{57}$$

Typical values with single precision are $h = 10^{-4}$ and $E = 2.10^{-4}$ and with double precision $h = 10^{-8}$ and $E = 2.10^{-8}$. These are the minimum error which are achievable with a forward difference numerical derivate.

To get a significant value of the step h, the step is computed with respect to the point where the derivate is to compute

$$h = r^{1/2}x \tag{58}$$

One can generalize the previous computation with the assumption that the scaling parameter from the Taylor expansion is h^{α_1} and the order of the formula is $\mathcal{O}(h^{\alpha_2})$. The total error is then

$$E = \frac{r}{h^{\alpha_1}} + h^{\alpha_2} \tag{59}$$

The optimal step is then

$$h = r^{\frac{1}{\alpha_1+\alpha_2}} \tag{60}$$

and the associated error is

$$E = 2r^{\frac{\alpha_2}{\alpha_1+\alpha_2}} \tag{61}$$

An additional trick [14] is to compute the step h so that the rounding error for the sum $x + h$ is minimum. This is performed by the following algorithm, which implies a temporary variable t

$$t = x + h \tag{62}$$
$$h = t - h \tag{63}$$

3.3.2 Results

In the following results, the variable x is either a scalar $x^i n\mathbb{R}$ or a vector $x \in \mathbb{R}^n$. When x is a vector, the step h_i is defined by

$$h_i = (0, \ldots, 0, 1, 0, \ldots, 0) \tag{64}$$

so that the only non-zero component of h_i is the i-th component.

- First derivate : forward 2 points

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{65}$$

  Optimal step : $h = r^{1/2}$ and error $E = 2r^{1/2}$.
  Single precision : $h \approx 10^{-4}$ and $E \approx 10^{-4}$.
  Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivate : backward 2 points

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h) \tag{66}$$

Optimal step : $h = r^{1/2}$ and error $E = 2r^{1/2}$.
Single precision : $h \approx 10^{-4}$ and $E \approx 10^{-4}$.
Double precision $h \approx 10^{-8}$ and $E \approx 10^{-8}$.

- First derivate : centered 2 points

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \tag{67}$$

Optimal step : $h = r^{1/3}$ and error $E = 2r^{2/3}$.
Single precision : $h \approx 10^{-3}$ and $E \approx 10^{-5}$.
Double precision $h \approx 10^{-5}$ and $E \approx 10^{-10}$.

3.3.3 Robust algorithm

The robust algorithm to compute the numerical derivate of a function of one variable is presented in figure 4.

$h \leftarrow \sqrt{\epsilon}$
$f'(x) \leftarrow (f(x+h) - f(x))/h$

Figure 4: A more robust algorithm to compute the numerical derivative of a function of one variable

3.4 One step further

As we can see, the *derivative* command is based on floating point comprehension. Is it completely *bulletproof* ? Not exactly.

See for example the following Scilab session.

```
-->fp = derivative(myfunction,1.e-100,order=1)
 fp  =
    0.000000014901161193847 7
-->fe=2.e-100
 fe  =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
 e  =
    7.450580596923828243D+91
```

The exact answer is $x_e = 2. \times 10^{-100}$ so that the result does not have any significant digits.

The additionnal experiment

The explanation is that the step is computed with $h = \sqrt{(eps)} \approx 10^{-8}$. Then $f(x+h) = f(10^{-100} + 10^{-8}) \approx f(10^{-8}) = 10^{-16}$, because the term 10^{-100} is much smaller than 10^{-8}. The result of the computation is therefore $(f(x+h) - f(x))/h = (10^{-16} + 10^{-200})/10^{-8} \approx 10^{-8}$.

The additionnal experiment

```
-->sqrt(%eps)
 ans  =
    0.00000001490116119384 77
```

explains the final result.

To improve the accuracy of the computation, one can take control of the step h. A reasonable solution is to use $h = \sqrt{\epsilon} * x$ so that the step is scaled depending on x. The following script illustrates than method, which produces results with 8 significant digits.

```
-->fp = derivative(myfunction,1.e-100,order=1,h=sqrt(%eps)*1.e-100)
 fp  =
    2.000000013099139394-100
-->fe=2.e-100
 fe  =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
 e  =
    0.000000006549569677 0794
```

But when x is exactly zero, the scaling method cannot work, because it would produce the step $h = 0$, and therefore a division by zero exception. In that case, the default step provides a good accuracy.

The *numdiff* command uses the step

$$h = \sqrt{\epsilon}(1 + 10^{-3}|x|) \tag{68}$$

As we can see the following session, the behaviour is approximately the same when the value of x is 1.

```
-->fp = numdiff(myfunction,1.0)
 fp  =
    2.000000018935341739 0237
-->fe=2.0
 fe  =
    2.
-->e = abs(fp-fe)/fe
 e  =
    9.468D-09
```

The accuracy is slightly decreased with respect to the optimal value 7.450581e-009 which was produced by derivative. But the number of significant digits is approximately the same, i.e. 9 digits.

The goal of this step is to produce good accuracy when the value of x is large, where the *numdiff* command produces accurate results, while *derivative* performs poorly.

```
-->numdiff(myfunction,1.e10)
  ans  =
    2.000D+10
```

```
-->derivative(myfunction,1.e10,order=1)
 ans  =
    0.
```

This step is a trade-off because it allows to keep a good accuracy with large values of x, but produces a slightly sub-optimal step size when x is near 1. The behaviour near zero is the same.

4 Complex division

Dans cette partie, nous analysons le problème de la division complexe dans Scilab. Nous mettons en lumière pourquoi le traitement numérique interne de Scilab est parfois inutilisé et pourquoi il est parfois inutile, voire nuisible (rappelons que le but d'une introduction est d'attirer le lecteur vers la suite du texte, raison pour laquelle nous avons rédigé ces mots les plus provocants possible !).

Nous détaillons en particulier la différence entre définition mathématique et implémentation en nombres flottants. Nous montrons comment la division de nombres complexes est effectué dans Scilab lorsque l'opérateur "/" est utilisé. Nous montrons également que l'implémentation n'est pas utilisé de manière consistante dans Scilab. Nous montrerons enfin que les librairies utilisées dans les compilateurs Intel et gfortran traitent le problème.

4.1 Theory

4.1.1 Algebraic computations

Il est de notoriété publique que la formule mathématique qui permet de calculer la division entre deux nombres complexes

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2} \tag{69}$$

4.1.2 Naive algorithm

The naive algorithm for the computation of the complex division is presented in figure 5.

$den \leftarrow c^2 + d^2$
$e \leftarrow (ac + bd)/den$
$f \leftarrow (bc - ad)/den$

Figure 5: Naive algorithm to compute the complex division

4.2 Experiments

n'est pas robuste quand on utilise des nombres flottants [18]. En résumé, le problème arrive lorsque les nombres a, b, c ou d s'approchent du domaine de définition des nombres flottants, ce qui peut provoquer un overflow ou un underflow.

Supposons que l'on traite la division suivante

$$\frac{1+I*1}{1+I*1e308} = 1e308 - I*1e-308 \tag{70}$$

Utilisons la formule mathématique, naïve, pour vérifier cette division.

$$\begin{aligned} den &= c+d = 1+(1e308) = 1+1e616\ 1e616 && (71)\\ e &= (ac+bd)/den = (1*1+1*1e308)/1e616\ 1e308/1e616\ 1e-308 && (72)\\ f &= (bc-ad)/den = (1*1-1*1e308)/1e616\ -1e308/1e616\ -1e-308 && (73) \end{aligned}$$

Numériquement, les choses ne se passent pas ainsi, car vmax = 1e308 est la valeur maximale avant overflow. Plus précisément, vmax est tel que tout nombre v > vmax satisfait

$$(v * 2)/2! = v \tag{74}$$

Autrement dit, la multiplication et la division ne sont plus précises pour des nombres v > vmax.

Si on utilise la formule naïve avec des nombres double precision, alors

$$den = c + d = 1 + (1e308) = Inf \tag{75}$$

c'est à dire un overflow. Les termes e et f sont alors calculés par

$$e = (ac + bd)/den = (1 * 1 + 1 * 1e308)/Inf = 1e308/Inf = 0 \tag{76}$$

$$f = (bc - ad)/den = (1 * 1 - 1 * 1e308)/Inf = -1e308/Inf = 0 \tag{77}$$

Le résultat est alors faux sur le plan mathématique, mais surtout, il est imprécis sur le plan numérique, dans la mesure où aucun des nombres initiaux n'était supérieur à vmax.

On peut également montrer que la formule naïve peut générer des underflow.

Supposons que l'on veuille calculer la division suivante :

$$\frac{1 + I * 1}{1e - 308 + I * 1e - 308} = 1e308 \tag{78}$$

Utilisons la formule mathématique, naïve, pour vérifier cette division.

$$den = c + d = (1e - 308) + (1e - 308) = 1e - 616 + 1e - 616 = 2e - 616 \tag{79}$$

$$e = (ac + bd)/den = (1 * 1e - 308 + 1 * 1e - 308)/2e - 616\ 2e - 308/2e - 616\ 1e308 \tag{80}$$

$$f = (bc - ad)/den = (1 * 1e - 308 - 1 * 1e - 308)/2e - 616\ 0/1e - 616\ 0 \tag{81}$$

Avec des nombres double precision, le calcul ne se passe pas ainsi. Le dénominateur va provoquer un underflow et va être numériquement mis à zéro, de telle sorte que

$$den = c + d = (1e - 308) + (1e - 308) = 1e - 616 + 1e - 616 = 0 \tag{82}$$

$$e = (ac + bd)/den = (1 * 1e - 308 + 1 * 1e - 308)/0\ 2e - 308/0\ Inf \tag{83}$$

$$f = (bc - ad)/den = (1 * 1e - 308 - 1 * 1e - 308)/0\ 0/0\ NaN \tag{84}$$

$$\tag{85}$$

4.2.1 Scilab implementation

On peut expérimenter facilement dans la console Scilab v5.0.2, prenant soin de choisir un cas test qui pose problème.

```
a=1+%i*1
b=1+%i*1e308
a/b
```

Le test montre que l'implémentation dans Scilab correspond à celle de Smith, puisque, dans ce cas, le résultat est correct :

```
-->a=1+%i*1
 a  =

    1. + i

-->b=1+%i*1e308
 b  =

    1. + 1.000+308i

-->a/b
 ans  =

    1.000-308 - 1.000-308i
```

Si on effectue le calcul pas à pas, on trouve que le code utilisé correspond au code source de "wwdiv" du module elementary_functions :

```
subroutine wwdiv(ar, ai, br, bi, cr, ci, ierr)
```

qui implémente la formule de Smith. Or, comme on l'a vu, l'algorithme de Smith possède une robustesse limitée.

On peut également tester les limites de la méthode de Smith, en expérimentant de la manière suivante :

```
-->a = 1e307 + %i * 1e-307
 a  =
    1.000+307 + 1.000-307i
-->b = 1e205 + %i * 1e-205
 b  =
    1.000+205 + 1.000-205i
-->a/b
 ans  =
    1.000+102
```

Cette réponse est fausse, mais correspond bien au résultat de la méthode de Smith.

Par ailleurs, il est intéressant de constater que la procédure wwdiv n'est pas utilisée systématiquement dans Scilab. En effet, on peut trouver des sections de code utilisant l'implémentation de la division complexe fournie par le compilateur.

Par exemple, la fonction lambda = spec(A,B) calcule les valeurs propres généralisées des matrices complexes A et B. L'interface vers la fonction zggev de Lapack est implémentée dans intzggev, dans laquelle on peut trouver les lignes suivantes

```
          do 15 i = 1, N
            zstk(lALPHA-1+i)=zstk(lALPHA-1+i)/zstk(lBETA-1+i)
 15       continue
```

Ce morceau de code permet de stocker dans le tableau complex associé à la variable alpha le résultat de la division de alpha/beta. Comme les deux opérandes sont de type complexe, c'est le compilateur fortran qui implémente la division (et, bien sûr, sans utiliser wwdiv).

4.2.2 Fortran code

Le code fortran suivant permet d'illustrer l'ensemble des points présentés précédement. On le test avec le compilateur Intel Fortran 10.1.

```
      program rndof
      complex*16 a
      complex*16 b
      complex*16 c
      double precision ar
      double precision ai
      double precision br
      double precision bi
      double precision cr
      double precision ci
c Check that naive implementation does not have a bug
      ar = 1.d0
      ai = 2.d0
      br = 3.d0
      bi = 4.d0
      call compare(ar, ai, br, bi)
c Check that naive implementation is not robust with respect to overflow
      ar = 1.d0
      ai = 1.d0
      br = 1.d0
      bi = 1.d308
      call compare(ar, ai, br, bi)
c Check that naive implementation is not robust with respect to underflow
      ar = 1.d0
      ai = 1.d0
      br = 1.d-308
      bi = 1.d-308
      call compare(ar, ai, br, bi)
c Check that Smith implementation is not robust with respect to complicated underflow
      ar = 1.d307
      ai = 1.d-307
      br = 1.d205
      bi = 1.d-205
      call compare(ar, ai, br, bi)
      end

      subroutine naive(ar, ai, br, bi, cr, ci)
      double precision, intent(in) :: ar
      double precision, intent(in) :: ai
      double precision, intent(in) :: br
      double precision, intent(in) :: bi
```

```
double precision, intent(out) :: cr
double precision, intent(out) :: ci
double precision den
den = br * br + bi * bi
cr = (ar * br + ai * bi) / den
ci = (ai * br - ar * bi) / den
end

subroutine smith(ar, ai, br, bi, cr, ci)
double precision, intent(in) :: ar
double precision, intent(in) :: ai
double precision, intent(in) :: br
double precision, intent(in) :: bi
double precision, intent(out) :: cr
double precision, intent(out) :: ci
double precision den
   if (abs(br) .ge. abs(bi)) then
      r = bi / br
      den = br + r*bi
      cr = (ar + ai*r) / den
      ci = (ai - ar*r) / den
   else
      r = br / bi
      den = bi + r*br
      cr = (ar*r + ai) / den
      ci = (ai*r - ar) / den
   endif
end

subroutine compare(ar, ai, br, bi)
double precision, intent(in) :: ar
double precision, intent(in) :: ai
double precision, intent(in) :: br
double precision, intent(in) :: bi
complex*16 a
complex*16 b
complex*16 c
double precision cr
double precision ci
print *, "****************"
call naive(ar, ai, br, bi, cr, ci)
print *, "Naive   :", cr, ci
call smith(ar, ai, br, bi, cr, ci)
print *, "Smith   :", cr, ci
a= dcmplx(ar, ai)
b = dcmplx(br, bi)
```

```
      c = a/b
      print * , "Fortran:", c
      end
```

Si on compile ce code avec Intel Fortran 10.1, on obtient l'affichage suivant dans la console.

```
****************
c naive:  0.440000000000000       8.000000000000000E-002
c Smith:  0.440000000000000       8.000000000000000E-002
c Fortran: (0.440000000000000,8.000000000000000E-002)
****************
c naive:  0.000000000000000E+000  0.000000000000000E+000
c Smith:  9.999999999999999E-309 -9.999999999999999E-309
c Fortran: (9.999999999999999E-309,-9.999999999999999E-309)
****************
c naive: Infinity                NaN
c Smith:  1.000000000000000E+308  0.000000000000000E+000
c Fortran: (1.000000000000000E+308,0.000000000000000E+000)
****************
c naive:  0.000000000000000E+000  0.000000000000000E+000
c Smith:  1.000000000000000E+102  0.000000000000000E+000
c Fortran: (9.999999999999999E+101,-9.999999999999999E-309)
```

Le quatrième test montre que l'implémentation fournie par le compilateur Intel donne un résultat correct, bien que la méthode de Smith donne de mauvais résultats.

4.2.3 C Code

Le code C++ suivant illustre le traitement du problème. Il est fondé sur le type "double complex" et fonctionne avec le compilateur Intel C 11.0 avec la configuration du standard C99 dans l'environnement Visual Studio.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

//
// naive --
//   Compute the complex division with a naive method.
//
void naive (double ar, double ai, double br, double bi, double * cr, double * ci)
{
double den;
den = br * br + bi * bi;
*cr = (ar * br + ai * bi) / den;
*ci = (ai * br - ar * bi) / den;
```

```
}

//
// smith --
//   Compute the complex division with Smith's method.
//
void smith (double ar, double ai, double br, double bi, double * cr, double * ci)
{
double den;
double r;
double abr;
double abi;
abr = fabs(br);
abi = fabs(bi);
if ( abr >= abi)
{
r = bi / br;
den = br + r*bi;
*cr = (ar + ai*r) / den;
*ci = (ai - ar*r) / den;
}
else
{
r = br / bi;
den = bi + r*br;
*cr = (ar*r + ai) / den;
*ci = (ai*r - ar) / den;
}
}

//
// compare --
//   Compare 3 methods for complex division:
//   * naive method
//   * Smith method
//   * C99 method
//
void compare (double ar, double ai, double br, double bi)
{
double complex a;
double complex b;
double complex c;

double cr;
double ci;
```

```
printf("****************\n");

naive(ar, ai, br, bi, &cr, &ci);
printf("Naif  --> c = %e +  %e * I\n" , cr ,  ci );

smith(ar, ai, br, bi, &cr, &ci);
printf("Smith --> c = %e +  %e * I\n" , cr ,  ci );

a = ar + ai*I;
b = br + bi*I;
c = a / b;
printf("C     --> c = %e +  %e * I\n" , creal(c) ,  cimag(c) );
}


int main(void)
{
double ar;
double ai;
double br;
double bi;


// Check that naive implementation does not have a bug
ar = 1;
ai = 2;
br = 3;
bi = 4;
compare (ar, ai, br, bi);

// Check that naive implementation is not robust with respect to overflow
ar = 1;
ai = 1;
br = 1;
bi = 1e307;
compare (ar, ai, br, bi);

// Check that naive implementation is not robust with respect to underflow
ar = 1;
ai = 1;
br = 1e-308;
bi = 1e-308;
compare (ar, ai, br, bi);

// Check that Smith method is not robust in complicated cases
ar = 1e307;
```

```
ai = 1e-307;
br = 1e205;
bi = 1e-205;
compare (ar, ai, br, bi);

return 0;
}
```

Voici le résultat qui apparaît dans la console :

```
****************
Naif  --> c = 4.400000e-001 +  8.000000e-002 * I
Smith --> c = 4.400000e-001 +  8.000000e-002 * I
C     --> c = 4.400000e-001 +  8.000000e-002 * I
****************
Naif  --> c = 0.000000e+000 +  -0.000000e+000 * I
Smith --> c = 1.000000e-307 +  -1.000000e-307 * I
C     --> c = 1.000000e-307 +  -1.000000e-307 * I
****************
Naif  --> c = 1.#INF00e+000 +  -1.#IND00e+000 * I
Smith --> c = 1.000000e+308 +  0.000000e+000 * I
C     --> c = 1.000000e+308 +  0.000000e+000 * I
****************
Naif  --> c = -1.#IND00e+000 +  -0.000000e+000 * I
Smith --> c = 1.000000e+102 +  0.000000e+000 * I
C     --> c = 1.000000e+102 +  -1.000000e-308 * I
```

Cela montre que l'implémentation des nombres complexes dans la librairie fournie par Intel traite le problème de manière adéquate.

4.3 Explanations

4.3.1 The Smith's method

C'est pourquoi les auteurs de Scilab, qui ont lu [7], ont implementé la formule de Smith [18] (mais ils citent Goldberg en référence, ce qui est une erreur) :

if $(|d| <= |c|)$ **then**
  $r \leftarrow d/c$
  $den \leftarrow c + r * d$
  $e \leftarrow (a + b * r)/den$
  $f \leftarrow (b - a * r)/den$
else
  $r \leftarrow c/d$
  $den \leftarrow d + r * c$
  $e \leftarrow (a * r + b)/den$
  $f \leftarrow (b * r - a)/den$
end if

Dans le cas $(1+i)/(1+1e308i)$, la méthode de Smith donne

```
si ( |1e308| <= |1| ) > test faux
sinon
  r = 1 / 1e308 = 0
  den  = 1e308 +  0 * 1 = 1e308
  e = (1 * 0 + 1) / 1e308 = 1e-308
  f = (1 * 0 - 1) / 1e308 = -1e-308
```

ce qui est le résultat correct.

Dans le cas $(1+i)/(1e-308+1e-308i)$, la méthode de Smith donne

```
si ( |1e-308| <= |1e-308| ) > test vrai
  r = 1e-308 / 1e308 = 1
  den  = 1e-308 +  1 * 1e-308 = 2e308
  e = (1 + 1 * 1) / 2e308 = 1e308
  f = (1 - 1 * 1) / 2e308 = 0
```

ce qui est encore une fois le résultat correct.

Il s'avère que le calcul de Smith, écrit en 1962, fonctionne bien dans un certain nombre de situations. L'article [4] cite une analyse de Hough qui donne une borne sur l'erreur réalisée par le calcul.

```
|zcomp - zref| <= eps |zref|
```

4.3.2 The limits of the Smith method

L'article [19] (1985) toutefois, fait la distinction entre la norme |zcomp - zref| et la valeur des parties imaginaires et réelles. Il montre en particulier un exemple dans lequel la partie imaginaire est erronée.

Supposons que m et n sont des entiers possédant les propriétés suivantes

```
m >> 0
n >> 0
n >> m
```

On peut alors facilement démontrer que la division complexe suivante peut être approchée :

```
10^n + i 10^-n
-------------- = 10^(n-m) - i 10^(n-3m)
10^m + i 10^-m
```

On décide alors de choisir les nombres n et m inférieurs à 308 mais de telle sorte que

```
n - 3 m = -308
```

Par exemple le couple m=205, n=307 satisfait les égalités précédentes de telle sorte que

```
10^307 + i 10^-307
------------------ = 10^102 - i 10^-308
10^205 + i 10^-205
```

Il est facile de voir que ce dernier cas met en défaut la formulation naïve. Il est plus surprenant de constater que ce cas met également en défaut la formule de Smith. En effet, les opérations suivantes sont réalisées par la méthode de Smith

```
si ( |1e-205| <= |1e205| ) > test vrai
  r = 1e-205 / 1e205 = 0
  den  = 1e205 +  0 * 1e-205 = 1e205
  e = (10^307 + 10^-307 * 0) / 1e205 = 1e102
  f = (10^-307 - 10^307 * 0) / 1e205 = 0
```

On constate que la partie réelle est exacte tandis que la partie imaginaire est fausse. On peut également vérifier que le module du résultat est dominé par la partie réelle de telle sorte que l'inégalité |zcomp - zref| <= eps |zref| reste vérifiée.

Les limites de la méthode de Smith ont étées levées dans [19]. L'algorithme proposé est fondé sur une proposition qui démontre que si n nombres x1...xn sont représentables alors min(xi) * max(xi) est également représentable. L'implémentation de la division complexe tire parti de cette proposition pour réaliser un calcul correct.

Il s'avère que l'algorithme de Stewart est dépassé par l'algorithme de Li et Al [12], mais également par celui de Kahan [10], qui, d'après [15], est similaire à celui implémenté dans le standard C99.

5 Conclusion

That article should not discourage us from implementing our own algorithms. Rather, it should warn us and that some specific event occur when we translate the mathematical material into a algorithm. That article shows us that accurate can be obtained with floating point numbers, provided that we are less *naïve*.

References

[1] Loss of significance. http://en.wikipedia.org/wiki/Loss_of_significance.

[2] Quadratic equation. http://en.wikipedia.org/wiki/Quadratic_equation.

[3] Quadratic equation. http://mathworld.wolfram.com/QuadraticEquation.html.

[4] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. 1972.

[5] George E. Forsythe. How do you solve a quadratic equation ? 1966. ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/66/40/CS-TR-66-40.pdf.

[6] George E. Forsythe. *How Do You Solve A Quadratic Equation ?* Computer Science Department, School of Humanities and Sciences, Stanford University, JUNE 1966. ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/66/40/CS-TR-66-40.pdf.

[7] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Association for Computing Machinery, Inc., March 1991. http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf.

[8] P. Dugac J. Dixmier. *Cours de Mathématiques du premier cycle, 1ère année*. Gauthier-Villars, 1969.

[9] M. A. Jenkins. Algorithm 493: Zeros of a real polynomial [c2]. *ACM Trans. Math. Softw.*, 1(2):178–189, 1975.

[10] W. KAHAN. Branch cuts for complex elementary functmns, or much ado about nothing's sign bit. pages 165–211, 1987.

[11] W. Kahan. On the cost of floating-point computation without extra-precise arithmetic. 2004. http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf.

[12] 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.

[13] Yves Nievergelt. How (not) to solve quadratic equations. *The College Mathematics Journal*, 34(2):90–104, 2003.

[14] W. H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, Second Edition*. 1992.

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

[16] K.E. Schmidt. Numerical derivatives. http://fermi.la.asu.edu/PHY531/intro/node1.html.

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

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

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