SCILAB IS NOT NAIVE

CONSORTIUM SCILAB
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex France

digiteo
Recherche en sciences & technologies de l'information

December 2010

Abstract

Most of the time, the mathematical formula is directly used in the Scilab source code. But, in many algorithms, some additional work is performed, which takes into account the fact that the computer does 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 naive and use algorithms which have been specifically tailored for floating point computers. We analyze in this article the particular case of the quadratic equation, the complex division and the numerical derivatives. In each example, we show that the naive algorithm is not sufficiently accurate, while Scilab implementation is much more robust.

Contents

1 Introduction

1.1 An open-source document

This document is an open-source project. The LaTeX sources are available on the Scilab Forge:

http://forge.scilab.org/index.php/p/docscilabisnotnaive/

The LaTeX sources are provided under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License:

http://creativecommons.org/licenses/by-sa/3.0

The Scilab scripts are provided on the Forge, inside the project, under the `scripts` sub-directory. The scripts are available under the CeCiLL licence:

http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt

1.2 Introduction

As a practical example of the problem considered in this document, consider the following numerical experiments. The following session is an example of a Scilab session, where we compute the real number 0.1 by two different, but mathematically equivalent ways.

```
-->format(25)
-->0.1
 ans  =
    0.1000000000000000055511
-->1.0-0.9
 ans  =
    0.0999999999999999777955
-->0.1 == 1.0 - 0.9
 ans  =
  F
```

I guess that for a person who has never heard of these problems, this experiment may be a shock. To get things clearer, let's check that the sinus function is also approximated in the sense that the value of $sin(\pi)$ is *not exactly* zero.

```
-->format(25)
-->sin(0.0)
 ans  =
    0.
-->sin(%pi)
 ans  =
    0.0000000000000001224647
```

With symbolic computation systems, such as Maple[32], Mathematica[41] or Maxima[2] for example, the calculations are performed with abstract mathematical symbols. Therefore, there is no loss of accuracy, as long as no numerical evaluation is performed. If a numerical solution is required as a rational number of the form p/q where p and q are integers and $q \neq 0$, there is still no loss of accuracy. On the other

hand, in numerical computing systems, such as Scilab[7], Matlab[33] or Octave[3] for example, the computations are performed with floating point numbers. When a numerical value is stored, it is generally associated with a rounding error.

The difficulty of numerical computations is generated by the fact that, while the mathematics treat with *real* numbers, the computer deals with their *floating point representations*. This is the difference between the *naive*, mathematical, approach, and the *numerical*, floating-point, implementation.

In this article, we will not present the floating point arithmetic in detail. Instead, we will show examples of floating point issues by using the following algebraic and experimental approach.

1. First, we will derive the basic theory of a mathematical formula.

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

3. Finally, we will analyze the *reasons* of the differences.

Our numerical experiments will be based on Scilab.

In order to measure the accuracy of the results, we will use two different measures of error: the relative error and the absolute error[20]. Assume that $x_c \in \mathbb{R}$ is a computed value and $x_e \in \mathbb{R}$ is the expected (exact) value. We are looking for a measure of the *difference* between these two real numbers. Most of the time, we use the relative error

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

where we assume that $x_e \neq 0$. The relative error e_r 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 instead the absolute error

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

A practical way of checking the expected result of a computation is to compare the formula computed "by hand" with the result produced by a symbolic tool. Recently, Wolfram has launched the http://www.wolframalpha.com website which let us access to Mathematica with a classical web browser. Many examples in this document have been validated with this tool.

In the following, we make a brief overview of floating point numbers used in Scilab. Real variables in Scilab are stored in *double precision* floating point variables. Indeed, Scilab uses the IEEE 754 standard so that real variables are stored with 64 bits floating point numbers, called *doubles*. The floating point number associated with a given $x \in \mathbb{R}$ will be denoted by $fl(x)$.

While the real numbers form a continuum, floating point numbers are both finite and bounded. Not all real numbers can be represented by a floating point number.

Indeed, there is a infinite number of reals, while there is a finite number of floating point numbers. In fact, there are, at most, 2^{64} different 64 bits floating point numbers. This leads to *roundoff*, *underflow* and *overflow*.

The double floating point numbers are associated with a machine epsilon equal to 2^{-52}, which is approximately equal to 10^{-16}. This parameter is stored in the `%eps` Scilab variable. Therefore, we can expect, at best, approximately 16 significant decimal digits. This parameter does not depend on the machine we use. Indeed, be it a Linux or a Windows system, Scilab uses IEEE doubles. Therefore, the value of the `%eps` variable is always the same in Scilab.

Negative normalized floating point numbers are in the range $[-10^{308}, -10^{-307}]$ and positive normalized floating point numbers are in the range $[10^{-307}, 10^{308}]$. The limits given in the previous intervals are only decimal approximations. Any real number greater than 10^{309} or smaller than -10^{309} is not representable as a double and is stored with the "infinite" value: in this case, we say that an overflow occurred. A real which magnitude is smaller than 10^{-324} is not representable as a double and is stored as a zero: in this case, we say that an underflow occurred.

The outline of this paper is the following. In the first section, we compute the roots of a quadratic equation. In the second section, we compute the numerical derivatives of a function. In the final section, we perform a numerically difficult division, with complex numbers. The examples presented in this introduction are presented in the appendix of this document.

2 Quadratic equation

In this section, we analyze the computation of the roots of a quadratic polynomial. As we shall see, there is a whole *world* from the mathematical 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 naive algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naive algorithm with the `roots` Scilab function. In the third part, we analyze why and how floating point numbers must be taken into account when the roots of a quadratic are required.

2.1 Theory

In this section, we present the mathematical formulas which allow to compute the real roots of a quadratic polynomial with real coefficients. We chose to begin by the example of a quadratic equation, because most of us exactly know (or *think* we know) how to solve such an equation with a computer.

Assume that $a, b, c \in \mathbb{R}$ are given coefficients and $a \neq 0$. Consider the following quadratic [4, 1, 5] equation:

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

where $x \in \mathbb{R}$ is the unknown.

Let us define by $\Delta = b^2 - 4ac$ the discriminant of the quadratic equation. We consider 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:

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

$$x_+ = \frac{-b + \sqrt{\Delta}}{2a}. \tag{5}$$

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

$$x_\pm = -\frac{b}{2a}. \tag{6}$$

- If $\Delta < 0$, there are two complex roots:

$$x_\pm = \frac{-b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a}. \tag{7}$$

We now consider a simplified algorithm where we only compute the real roots of the quadratic, assuming that $\Delta > 0$. This naive algorithm is presented in figure 1.

input : a, b, c
output: x_-, x_+
$\Delta := b^2 - 4ac$;
$s := \sqrt{\Delta}$;
$x_- := (-b - s)/(2a)$;
$x_+ := (-b + s)/(2a)$;

Algorithm 1: Naive algorithm to compute the real roots of a quadratic equation. - We assume that $\Delta > 0$.

2.2 Experiments

In this section, we compare our naive algorithm with the `roots` function. We begin by defining a function which naively implements the mathematical formulas. Then we use our naive function on two particular examples. In the first example, we focus on massive cancellation and in the second example, we focus on overflow problems.

The following Scilab function `myroots` is a straightforward implementation of the previous formulas. It takes as input the coefficients of the quadratic, stored in the vector variable `p`, and returns the two roots in the vector `r`.

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

2.2.1 Massive cancellation

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

$$\epsilon x^2 + (1/\epsilon)x - \epsilon = 0 \tag{8}$$

with $\epsilon > 0$. The discriminant of this equation is $\Delta = 1/\epsilon^2 + 4\epsilon^2$. The two real solutions of the quadratic equation are

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

We are mainly interested in the case where the magnitude of ϵ is very small. The roots are approximated by

$$x_- \approx -1/\epsilon^2, \qquad x_+ \approx \epsilon^2, \tag{10}$$

when ϵ is close to zero. We now consider the limit of the two roots when $\epsilon \to 0$. We have

$$\lim_{\epsilon \to 0} x_- = -\infty, \qquad \lim_{\epsilon \to 0} x_+ = 0. \tag{11}$$

In the following Scilab script, we compare the roots computed by the `roots` function and the roots computed by our naive function. Only the positive root $x_+ \approx \epsilon^2$ is considered in this test. Indeed, the x_- root is so that $x_- \to -\infty$ in both implementations. We consider the special case $\epsilon = 0.0001 = 10^{-4}$. We begin by creating a polynomial with the `poly` function, which is given the coefficients of the polynomial. The variable `e1` contains the expected value of the positive root $x_+ = \epsilon^2$. Then we compute the roots `r1` and `r2` with the two functions `roots` and `myroots`. We finally compute the relative errors `error1` and `error2`.

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

The previous script produces the following output.

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

We see that the naive method produces a root which has no significant digit and a relative error which is 14 orders of magnitude greater than the relative error of the Scilab root.

This behavior is explained by the fact that the expression for the positive root x_+ given by the equality 5 is numerically evaluated as following. We first consider how the discriminant $\Delta = 1/\epsilon^2 + 4\epsilon^2$ is computed. The term $1/\epsilon^2$ is equal to 100000000 and the term $4\epsilon^2$ is equal to 0.00000004. Therefore, the sum of these two terms is equal to 100000000.000000045. Hence, the square root of the discriminant is

$$\sqrt{1/\epsilon^2 + 4\epsilon^2} = 10000.000000000001818989. \tag{12}$$

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

$$\begin{aligned} -1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2} &= -10000.0 + 10000.000000000001818989 & (13)\\ &= 0.00000000000018189894035 & (14) \end{aligned}$$

We see that the result is mainly driven by the cancellation of significant digits.

We may think that the result is extreme, but it is not. For example, consider the case where we reduce further 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. There is no significant decimal digit in the result. 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 orders of magnitude greater than the relative error produced by the `roots` function.

2.2.2 Overflow

In this section, we analyse the overflow which appears when the discriminant of the quadratic equation is such that $b^2 - 4ac$ is not representable as a double. We consider the following quadratic equation

$$x^2 + (1/\epsilon)x + 1 = 0 \tag{15}$$

with $\epsilon > 0$. We especially consider the case $\epsilon \to 0$. The discriminant of this equation is $\Delta = 1/\epsilon^2 - 4$. Assume that the discriminant is positive. Therefore, the roots of the quadratic equation are

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 - 4}}{2}, \qquad x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 - 4}}{2}. \tag{16}$$

These roots are approximated by

$$x_- \approx -1/\epsilon, \qquad x_+ \approx -\epsilon, \tag{17}$$

when ϵ is close to zero. We now consider the limit of the two roots when $\epsilon \to 0$. We have

$$\lim_{\epsilon \to 0} x_- = -\infty, \qquad \lim_{\epsilon \to 0} x_+ = 0^-. \tag{18}$$

To create a difficult case, we search ϵ so that $1/\epsilon^2 > 10^{308}$, because we know that 10^{308} is the maximum representable double precision floating point number. Therefore, we expect that something should go wrong in the computation of the expression $\sqrt{1/\epsilon^2 - 4}$. We choose $\epsilon = 10^{-155}$.

In the following script, we compare the roots computed by the `roots` function and our naive implementation.

```
e=1.e-155
a = 1;
b = 1/e;
c = 1;
p=poly([c b a],"x","coeff");
expected = [-e;-1/e];
roots1 = myroots(p);
roots2 = roots(p);
error1 = abs(roots1-expected)/norm(expected);
error2 = abs(roots2-expected)/norm(expected);
printf("Expected : %e %e\n", expected(1),expected(2));
printf("Naive method : %e %e (error=%e %e)\n", ...
  roots1(1),roots1(2), error1(1),error1(2));
printf("Scilab method : %e %e (error=%e %e)\n", ...
  roots2(1),roots2(2), error2(1),error2(2));
```

The previous script produces the following output.

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

In this case, the discriminant $\Delta = b^2 - 4ac$ has been evaluated as $1/\epsilon^2 - 4$, which is approximately equal to 10^{310}. This number cannot be represented in a double precision floating point number. It therefore produces the IEEE Infinite number, which is displayed by Scilab as `Inf`. The Infinite number is associated with an algebra and functions can perfectly take this number as input. Therefore, when the square root function must compute $\sqrt{\Delta}$, it produces again `Inf`. This number is then propagated into the final roots.

2.3 Explanations

In this section, we suggest robust methods to compute the roots of a quadratic equation.

The methods presented in this section are extracted from the *quad* routine of the *RPOLY* algorithm by Jenkins and Traub [24, 23]. This algorithm is used by Scilab in the `roots` function, where a special case is used when the degree of the equation is equal to 2, i.e. a quadratic equation.

2.3.1 Properties of the roots

In this section, we present elementary results, which will be used in the derivation of robust floating point formulas of the roots of the quadratic equation.

Let us assume that the quadratic equation 3, with real coefficients $a, b, c \in \mathbb{R}$ and $a > 0$ has a positive discriminant $\Delta = b^2 - 4ac$. Therefore, the two real roots of

the quadratic equation are given by the equations 4 and 5. We can prove that the sum and the product of the roots satisfy the equations

$$x_- + x_+ = \frac{-b}{a}, \qquad x_- x_+ = \frac{c}{a}. \tag{19}$$

Therefore, the roots are the solution of the normalized quadratic equation

$$x^2 - (x_- + x_+)x + x_- x_+ = 0. \tag{20}$$

Another transformation leads to an alternative form of the roots. Indeed, the original quadratic equation can be written as a quadratic equation of the unknown $1/x$. Consider the quadratic equation 3 and divide it by $1/x^2$, assuming that $x \neq 0$. This leads to the equation

$$c(1/x)^2 + b(1/x) + a = 0, \tag{21}$$

where we assume that $x \neq 0$. The two real roots of the quadratic equation 21 are

$$x_- = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{22}$$

$$x_+ = \frac{2c}{-b - \sqrt{b^2 - 4ac}}. \tag{23}$$

The expressions 22 and 23 can also be derived directly from the equations 4 and 5. For that purpose, it suffices to multiply their numerator and denominator by $-b + \sqrt{b^2 - 4ac}$.

2.3.2 Floating-Point implementation : overview

The numerical experiments presented in sections 2.2.1 and 2.2.2 suggest that the floating point implementation must deal with two different problems:

- massive cancellation when $b^2 \gg 4ac$ because of the cancellation of the terms $-b$ and $\pm\sqrt{b^2 - 4ac}$ which may have opposite signs,
- overflow in the computation of the square root of the discriminant $\sqrt{\pm(b^2 - 4ac)}$ when $b^2 - 4ac$ is not representable as a floating point number.

The cancellation problem occurs only when the discriminant is positive, i.e. only when there are two real roots. Indeed, the cancellation will not appear when $\Delta < 0$, since the complex roots do not use the sum $-b \pm \sqrt{b^2 - 4ac}$. When $\Delta = 0$, the double real root does not cause any trouble. Therefore, we must take into account for the cancellation problem only in the equations 4 and 5.

On the other hand, the overflow problem occurs whatever the sign of the discriminant but does not occur when $\Delta = 0$. Therefore, we must take into account for this problem in the equations 4, 5 and 7. In section 2.3.3, we focus on the cancellation error while the overflow problem is addressed in section 2.3.4.

2.3.3 Floating-Point implementation : fixing massive cancellation

In this section, we present the computation of the roots of a quadratic equation with protection against massive cancellation.

When the discriminant Δ is positive, the massive cancellation problem can be split in two cases:

- if $b < 0$, then $-b - \sqrt{b^2 - 4ac}$ may suffer of massive cancellation because $-b$ is positive and $-\sqrt{b^2 - 4ac}$ is negative,
- if $b > 0$, then $-b + \sqrt{b^2 - 4ac}$ may suffer of massive cancellation because $-b$ is negative and $\sqrt{b^2 - 4ac}$ is positive.

Therefore,

- if $b > 0$, we should use the expression $-b - \sqrt{b^2 - 4ac}$,
- if $b < 0$, we should use the expression $-b + \sqrt{b^2 - 4ac}$.

The solution consists in a combination of the following expressions of the roots given by, on one hand the equations 4 and 5, and, on the other hand the equations 22 and 23. We 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 massive cancellation problem:

- if $b < 0$, then compute x_- from 22, else (if $b > 0$), compute x_- from 4,
- if $b < 0$, then compute x_+ from 5, else (if $b > 0$), compute x_+ from 23.

We can also consider the modified Fagnano formulas

$$x_1 = -\frac{2c}{b + sgn(b)\sqrt{b^2 - 4ac}}, \tag{24}$$

$$x_2 = -\frac{b + sgn(b)\sqrt{b^2 - 4ac}}{2a}, \tag{25}$$

where the sign function is defined by

$$sgn(b) = \begin{cases} 1, & \text{if } b \geq 0, \\ -1, & \text{if } b < 0. \end{cases} \tag{26}$$

The roots $x_{1,2}$ correspond to the roots $x_{+,-}$. Indeed, on one hand, 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 $b > 0$, $x_2 = x_-$.

Moreover, we notice 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

$$x_1 = -\frac{c}{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}, \tag{27}$$

$$x_2 = -\frac{(b/2) + sgn(b)\sqrt{(b/2)^2 - ac}}{a}. \tag{28}$$

Therefore, the two real roots can be computed by the following sequence of computations:

$$b' := b/2, \qquad \Delta' := b'^2 - ac, \tag{29}$$

$$h := -\left(b' + sgn(b)\sqrt{\Delta'}\right) \tag{30}$$

$$x_1 := \frac{c}{h}, \qquad x_2 := \frac{h}{a}. \tag{31}$$

In the case where the discriminant $\Delta' := b'^2 - ac$ is negative, the two complex roots are

$$x_1 = -\frac{b'}{a} - i\frac{\sqrt{ac - b'^2}}{a}, \qquad x_2 = -\frac{b'}{a} + i\frac{\sqrt{ac - b'^2}}{a}. \tag{32}$$

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 the square root of the discriminant $\sqrt{\pm(b'^2 - ac)}$ without creating unnecessary overflows. In order to simplify the discussion, we focus on the computation of $\sqrt{b'^2 - ac}$.

Obviously, the problem occur for large values of b'. Notice that a (very) small improvement has already been done. Indeed, we have the inequality $|b'| = |b|/2 < |b|$ so that overflows are twice less likely to occur. The current upper bound for $|b'|$ is 10^{154}, which is associated with $b'^2 \leq 10^{308}$, the maximum double value before overflow. The goal is therefore to increase the possible range of values of b' without generating unnecessary overflows.

Consider the situation when b' is large in magnitude with respect to a and c. In that case, notice that we first square b' to get b'^2 and then compute the square root $\sqrt{b'^2 - ac}$. Hence, we can factor the expression by b'^2 and move this term outside the square root, which makes the term $|b'|$ appear. This method allows to compute the expression $\sqrt{b'^2 - ac}$, without squaring b' when it is not necessary.

In the general case, we use the fact that the term $b'^2 - ac$ can be evaluated with the two following equivalent formulas:

$$b'^2 - ac = b'^2\left[1 - (a/b')(c/b')\right], \tag{33}$$

$$b'^2 - ac = c\left[b'(b'/c) - a\right]. \tag{34}$$

The goal is then to compute the square root $s = \sqrt{b'^2 - ac}$.

- If $|b'| > |c| > 0$, then the equation 33 involves the expression $1 - (a/b')(c/b')$. The term $1 - (a/b')(c/b')$ is so that no overflow is possible since $|c/b'| < 1$ (the overflow problem occurs only when b is large in magnitude with respect to both a and c). In this case, we use the expression

$$e = 1 - (a/b')(c/b'), \tag{35}$$

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

Algorithm 2: A more robust algorithm to compute the roots of a quadratic equation. This algorithm takes as input arguments the real coefficients a, b, c and returns the real and imaginary parts of the two roots, i.e. returns x_-^R, x_-^I, x_+^R, x_+^I.

and compute

$$s = \pm|b'|\sqrt{|e|}. \tag{36}$$

In the previous equation, we use the sign + when e is positive and the sign - when e is negative.

- If $|c| > |b'| > 0$, then the equation 34 involves the expression $b'(b'/c) - a$. The term $b'(b'/c) - a$ should limit the possible overflows since $|b'/c| < 1$. This implies that $|b'(b'/c)| < |b'|$. (There is still a possibility of overflow, for example in the case where $b'(b'/c)$ is near, but under, the overflow limit and a is large.) Therefore, we use the expression

$$e = b'(b'/c) - a, \tag{37}$$

and compute

$$s = \pm\sqrt{|c|}\sqrt{|e|}. \tag{38}$$

In the previous equation, we use the sign + when e is positive and the sign - when e is negative.

In both equations 36 and 38, the parenthesis must be strictly used. This property is ensured by the IEEE standard and by the Scilab language. This normalization method are similar to the one used by Smith in the algorithm for the division of complex numbers [44] and which will be reviewed in the next section.

2.3.5 Conditioning of the problem

The conditioning 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 by associated with the coefficients of the quadratic equations. In the following, we assume that $a \neq 0$.

First, assume that the discriminant is positive, i.e. assume that $\Delta > 0$. Therefore, the roots given by the equations 4 and 5 can be directly differentiated. This leads to

$$\begin{array}{ll} \frac{\partial x_-}{\partial a} = \frac{c}{a\sqrt{\Delta}} + \frac{b+\sqrt{\Delta}}{2a^2}, & \frac{\partial x_+}{\partial a} = -\frac{c}{a\sqrt{\Delta}} + \frac{b-\sqrt{\Delta}}{2a^2} \\ \frac{\partial x_-}{\partial b} = \frac{-1-b/\sqrt{\Delta}}{2a}, & \frac{\partial x_+}{\partial b} = \frac{-1+b/\sqrt{\Delta}}{2a} \\ \frac{\partial x_-}{\partial c} = \frac{1}{\sqrt{\Delta}}, & \frac{\partial x_+}{\partial c} = -\frac{1}{\sqrt{\Delta}}. \end{array} \tag{39}$$

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

$$\frac{\partial x_\pm}{\partial a} = \frac{b}{2a^2}, \qquad \frac{\partial x_\pm}{\partial b} = \frac{-1}{2a}, \qquad \frac{\partial x_\pm}{\partial c} = 0. \tag{40}$$

In both cases, we see that when the coefficient a converges toward zero, some of the partial derivatives are converging toward $\pm\infty$. For example, if $\Delta > 0$, then

$\lim_{a\to 0} \frac{\partial x_-}{\partial a} \to \pm\infty$. We also see that behavior when the discriminant converges toward zero. This implies that, in the case where a or Δ are small, a small change in the coefficient is associated to a large change in the roots. This is the definition of an ill-conditioned problem.

We can relate this ill-conditioning problem to the eigenvalue problem of a square matrix A. Indeed, it is a well-known result that, when A is non-normal, it may happen that small changes in A can induce large changes in the eigenvalues[19], chapter 7, "The Unsymmetric Eigenvalue Problem". The eigenvalue problem and the roots of a quadratic equation are related by the fact that the eigenvalues are the roots of the characteristic polynomial. Moreover, a very general method to find the roots of a polynomial is to find the eigenvalues of its companion matrix.

2.4 References

The 1966 technical report by G. Forsythe [12] presents the floating point system and the possible large error in using mathematical algorithms blindly. An accurate way of solving a quadratic is outlined. A few general remarks are made about computational mathematics.

The 1991 paper by Goldberg [18] is a general presentation of the floating point system and its consequences. It begins with background on floating point representation and rounding errors, continues with a discussion of the IEEE floating point standard and concludes with examples of how computer system builders can better support floating point. The section 1.4, "Cancellation" specifically consider the computation of the roots of a quadratic equation.

We can also read the numerical experiments performed by Nievergelt in [38].

The Numerical Recipes [39], chapter 5, section 5.6, "Quadratic and Cubic Equations" present the elementary theory for a floating point implementation of the quadratic and cubic equations.

Other references include William Kahan [26].

2.5 Exercises

Exercise 2.1 (*Roots of the normalized quadratic equation*) We consider the normalized quadratic equation:

$$x^2 + ax + b = 0, \tag{41}$$

where $a, b \in \mathbb{R}$ are given coefficients and $x \in \mathbb{R}$ is the unknown. Prove that the roots of the normalized quadratic equation 41 are the following. Let us define $\delta = \frac{a^2}{4} - b$.

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

$$x_\pm = -\frac{a}{2} \pm \sqrt{\frac{a^2}{4} - b} \tag{42}$$

- If $\delta = 0$, there is one double real root,

$$x = -\frac{a}{2}. \tag{43}$$

- If $\delta < 0$, there are two complex roots,

$$x_\pm = -\frac{a}{2} \pm i\sqrt{b - \frac{a^2}{4}} \tag{44}$$

Exercise 2.2 (*Roots of the quadratic equation*) We consider the quadratic equation 3 where $a, b, c \in \mathbb{R}$ are given coefficients and $x \in \mathbb{R}$ is the unknown. We assume that $a \neq 0$. Prove that the roots of the quadratic equation 3 are given by the equations 4, 5, 6 and 7.

Exercise 2.3 (*Properties of the roots*) Prove the equations 19.

Exercise 2.4 (*Inverted roots*) Based on the equations 4 and 5, prove directly the equations 22 and 23, i.e. prove that

$$x_- = \frac{-b-\sqrt{\Delta}}{2a} = \frac{2c}{-b+\sqrt{b^2-4ac}}, \tag{45}$$

$$x_+ = \frac{-b+\sqrt{\Delta}}{2a} = \frac{2c}{-b-\sqrt{b^2-4ac}}. \tag{46}$$

Exercise 2.5 (*Expansion of $\sqrt{1+x}$ near $x=0$*) Prove that, when x is in the neighborhood of zero, we have

$$\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3 + O(x^4). \tag{47}$$

Exercise 2.6 (*Roots of a quadratic equation #1*) Prove the approximations 10.

Exercise 2.7 (*Roots of a quadratic equation #2*) Prove the approximations 17.

2.6 Answers to exercises

Answer of Exercise 2.1 (*Roots of the normalized quadratic equation*) We use the following change of variable

$$x = t + \lambda \tag{48}$$

where $t \in \mathbb{R}$ is the unknown and $\lambda \in \mathbb{R}$ is a parameter. This parameter will be tuned so that the linear term in the quadratic equation is zero. We plug the change of variable 48 into the equation 41 and get

$$x^2 + ax + b = (t+\lambda)^2 + a(t+\lambda) + b \tag{49}$$

$$= t^2 + 2t\lambda + \lambda^2 + at + a\lambda + b \tag{50}$$

$$= 0. \tag{51}$$

We can organize the previous expression by decreasing powers of the unknown t. Hence, the unknown $t \in \mathbb{R}$ must be a solution of the equation

$$t^2 + (a+2\lambda)t + (\lambda^2 + a\lambda + b) = 0 \tag{52}$$

The linear term is zero if

$$a + 2\lambda = 0. \tag{53}$$

Therefore, we choose to define λ by the equation

$$\lambda = -\frac{a}{2}. \tag{54}$$

Hence, the change of variable is

$$x = t - \frac{a}{2}. \tag{55}$$

The constant term in the equation 52 can be simplified into

$$\lambda^2 + a\lambda + b = \left(-\frac{a}{2}\right)^2 + a\left(-\frac{a}{2}\right) + b \tag{56}$$

$$= \frac{a^2}{4} - \frac{a^2}{2} + b \tag{57}$$

$$= -\frac{a^2}{4} + b. \tag{58}$$

The equation 52 is now simplified and the unknown t must be a root of the equation

$$t^2 + (b - \frac{a^2}{4}) = 0. \tag{59}$$

The previous equation can be expressed as

$$t^2 = \delta \tag{60}$$

where δ is defined by

$$\delta = \frac{a^2}{4} - b. \tag{61}$$

The roots of the quadratic equation 60 can be found depending on the sign of δ.

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

$$t_\pm = \pm\sqrt{\delta}. \tag{62}$$

The roots $x_\pm$ can then be computed by using the change of variable 55. This leads to

$$\begin{aligned} x_\pm &= t_\pm - \frac{a}{2} && (63)\\ &= -\frac{a}{2} \pm \sqrt{\delta} && (64)\\ &= -\frac{a}{2} \pm \sqrt{\frac{a^2}{4} - b} && (65) \end{aligned}$$

- If $\delta = 0$, there is one double root

$$t_\pm = 0. \tag{66}$$

The change of variable 55 leads to

$$x_\pm = -\frac{a}{2}. \tag{67}$$

- If $\delta < 0$, there are two complex roots,

$$t_\pm = \pm i\sqrt{-\delta} \tag{68}$$

and the change of variable 55 leads to

$$\begin{aligned} x_\pm &= -\frac{a}{2} \pm i\sqrt{-\delta} && (69)\\ &= -\frac{a}{2} \pm i\sqrt{b - \frac{a^2}{4}}. && (70) \end{aligned}$$

We have analyzed the roots depending on the sign of the discriminant δ and the proof is complete. □

Answer of Exercise 2.2 (*Roots of the quadratic equation*) We use the result of the exercise 2.1. We consider the quadratic equation 3 where $a, b, c \in \mathbb{R}$ are given coefficients and $x \in \mathbb{R}$ is the unknown. We assume that $a \neq 0$. Therefore, we can divide the equation 3 by a and get

$$x^2 + a'x + b' = 0, \tag{71}$$

where $a' = \frac{b}{a}$ and $b' = \frac{c}{a}$. The discriminant is

$$\begin{aligned} \delta &= \frac{a'^2}{4} - b' && (72)\\ &= \frac{(b/a)^2}{4} - (c/a) && (73)\\ &= \frac{b^2}{4a^2} - \frac{c}{a} && (74)\\ &= \frac{b^2 - 4ac}{4a^2}. && (75) \end{aligned}$$

We define Δ by the equation $\Delta = b^2 - 4ac$. Therefore, we have $\delta = \frac{\Delta}{4a^2}$. Since $4a^2 > 0$, the signs of δ and Δ are the same and $\delta = 0$ if and only if $\Delta = 0$.

We plug the previous definitions of a' and b' into the roots of the normalized equation given by exercise 2.1 and get the following result.

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

$$\begin{aligned} x_\pm &= -\frac{a'}{2} \pm \sqrt{\delta} && (76)\\ &= -\frac{b}{2a} \pm \frac{\sqrt{\Delta}}{2a} && (77)\\ &= \frac{-b \pm \sqrt{\Delta}}{2a} && (78) \end{aligned}$$

  which proves the equations 4 and 5.

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

$$\begin{aligned} x_\pm &= -\frac{a'}{2} && (79)\\ &= -\frac{b}{2a}. && (80) \end{aligned}$$

  We have proved the equation 6.

- If $\Delta < 0$, there are two complex roots:

$$\begin{aligned} x_\pm &= -\frac{a'}{2} \pm i\sqrt{-\delta} && (81)\\ &= -\frac{b}{2a} \pm i\frac{\sqrt{-\Delta}}{2a} && (82) \end{aligned}$$

  which proves the equation 7.

□

Answer of Exercise 2.3 (*Properties of the roots*) Let us prove the equations 19 in the three cases $\Delta > 0$, $\Delta = 0$ and $\Delta < 0$. First, assume that $\Delta > 0$. Then, by the equations 4 and 5, the sum of the roots is

$$\begin{aligned} x_- + x_+ &= \frac{-b - \sqrt{\Delta}}{2a} + \frac{-b + \sqrt{\Delta}}{2a} && (83)\\ &= \frac{-2b}{2a} && (84)\\ &= \frac{-b}{a}. && (85) \end{aligned}$$

The product of the roots is

$$\begin{aligned} x_-.x_+ &= \left(\frac{-b - \sqrt{\Delta}}{2a}\right)\left(\frac{-b + \sqrt{\Delta}}{2a}\right) && (86)\\ &= \frac{b^2 - \Delta}{4a^2} && (87)\\ &= \frac{b^2 - (b^2 - 4ac)}{4a^2} && (88)\\ &= \frac{4ac}{4a^2} && (89)\\ &= \frac{c}{a}. && (90) \end{aligned}$$

Second, assume that $\Delta = 0$. By the equation 6, the sum of the roots is

$$x_- + x_+ = \frac{-b}{2a} + \frac{-b}{2a} \tag{91}$$

$$= \frac{-b}{a}. \tag{92}$$

The product of the roots is

$$x_-.x_+ = \left(\frac{-b}{2a}\right)\left(\frac{-b}{2a}\right) \tag{93}$$

$$= \frac{b^2}{4a^2}. \tag{94}$$

But the equality $\Delta = 0$ implies $b^2 = 4ac$. We plug this last equality in the equation 94, and we find

$$x_-.x_+ = \frac{4ac}{4a^2} \tag{95}$$

$$= \frac{c}{a}. \tag{96}$$

Finally, assume that $\Delta < 0$. Then, by the equations 7, the sum of the roots is

$$x_- + x_+ = \frac{-b}{2a} - i\frac{\sqrt{-\Delta}}{2a} + \frac{-b}{2a} + i\frac{\sqrt{-\Delta}}{2a} \tag{97}$$

$$= \frac{-2b}{2a} \tag{98}$$

$$= \frac{-b}{a}. \tag{99}$$

The product of the roots is

$$x_-.x_+ = \left(\frac{-b}{2a} - i\frac{\sqrt{-\Delta}}{2a}\right)\left(\frac{-b}{2a} + i\frac{\sqrt{-\Delta}}{2a}\right) \tag{100}$$

$$= \left(\frac{-b}{2a}\right)^2 + \left(\frac{\sqrt{-\Delta}}{2a}\right)^2 \tag{101}$$

$$= \frac{b^2}{4a^2} + \frac{-\Delta}{4a^2} \tag{102}$$

$$= \frac{b^2}{4a^2} + \frac{4ac - b^2}{4a^2} \tag{103}$$

$$= \frac{4ac}{4a^2} \tag{104}$$

$$= \frac{c}{a}. \tag{105}$$

The three cases $\Delta > 0$, $\Delta = 0$ and $\Delta < 0$ have been considered so that the proof is complete. □

Answer of Exercise 2.4 (*Inverted roots*) Based on the equations 4 and 5, let us prove directly the equations 22 and 23, i.e. let us prove that

$$x_- = \frac{-b - \sqrt{\Delta}}{2a} = \frac{2c}{-b + \sqrt{b^2 - 4ac}}, \tag{106}$$

$$x_+ = \frac{-b + \sqrt{\Delta}}{2a} = \frac{2c}{-b - \sqrt{b^2 - 4ac}}. \tag{107}$$

First, we multiply the numerator and the denominator of x_- by $-b+\sqrt{\Delta}$. This leads to

$$\begin{aligned} x_- &= \frac{-b-\sqrt{\Delta}}{2a}\cdot\frac{-b+\sqrt{\Delta}}{-b+\sqrt{\Delta}} && (108)\\ &= \frac{(-b)^2-(\sqrt{\Delta})^2}{2a(-b+\sqrt{\Delta})} && (109)\\ &= \frac{b^2-\Delta}{2a(-b+\sqrt{\Delta})} && (110)\\ &= \frac{b^2-(b^2-4ac)}{2a(-b+\sqrt{\Delta})} && (111)\\ &= \frac{4ac}{2a(-b+\sqrt{\Delta})} && (112)\\ &= \frac{2c}{-b+\sqrt{\Delta}}. && (113) \end{aligned}$$

Second, we multiply the numerator and the denominator of x_+ by $-b-\sqrt{\Delta}$. This leads to

$$\begin{aligned} x_+ &= \frac{-b+\sqrt{\Delta}}{2a}\cdot\frac{-b-\sqrt{\Delta}}{-b-\sqrt{\Delta}} && (114)\\ &= \frac{(-b)^2-(\sqrt{\Delta})^2}{2a(-b-\sqrt{\Delta})} && (115)\\ &= \frac{b^2-\Delta}{2a(-b-\sqrt{\Delta})} && (116)\\ &= \frac{b^2-(b^2-4ac)}{2a(-b-\sqrt{\Delta})} && (117)\\ &= \frac{4ac}{2a(-b-\sqrt{\Delta})} && (118)\\ &= \frac{2c}{-b-\sqrt{\Delta}}. && (119) \end{aligned}$$

This proves the equalities 22 and 23. □

Answer of Exercise 2.5 (*Expansion of* $\sqrt{1+x}$ *near* $x=0$) Assume that f is a continuously differentiable function. By Taylor's theorem, we have

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4). \quad (120)$$

We use the Taylor's expansion 120 with $f(x)=\sqrt{x}$ in the neighborhood of $x=1$. The derivatives of the function f are

$$f'(x) = \frac{1}{2}x^{-\frac{1}{2}}, \qquad f''(x) = -\frac{1}{4}x^{-\frac{3}{2}}, \qquad f'''(x) = \frac{3}{8}x^{-\frac{5}{2}}, \quad (121)$$

so that

$$f(1)=1, \qquad f'(1)=\frac{1}{2}, \qquad f''(1)=-\frac{1}{4}, \qquad f'''(1) = \frac{3}{8}. \quad (122)$$

The Taylor expansion 120 therefore implies

$$\begin{aligned} \sqrt{1+h} &= 1 + h\cdot\frac{1}{2} + \frac{1}{2}h^2\cdot\left(-\frac{1}{4}\right) + \frac{1}{6}h^3\cdot\frac{3}{8} + O(h^4), && (123)\\ &= 1 + \frac{1}{2}h - \frac{1}{8}h^2 + \frac{1}{16}h^3 + O(h^4). && (124) \end{aligned}$$

The previous equality proves the equality 47 and concludes the proof. □

Answer of Exercise 2.6 (*Roots of a quadratic equation #1*) Let us prove the approximations 10. The two real solutions of the quadratic equation 8 are

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon}, \qquad x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2}}{2\epsilon}. \tag{125}$$

Let us prove that

$$x_- \approx -1/\epsilon^2, \qquad x_+ \approx \epsilon^2. \tag{126}$$

Based on the Taylor expansion 47, we can simplify the expression $\sqrt{1/\epsilon^2 + 4\epsilon^2}$. Indeed, we have

$$\begin{aligned} \sqrt{1/\epsilon^2 + 4\epsilon^2} &= \sqrt{\frac{1 + 4\epsilon^4}{\epsilon^2}} && (127)\\ &= \frac{\sqrt{1 + 4\epsilon^4}}{\epsilon} && (128) \end{aligned}$$

By the equation 47, we have

$$\begin{aligned} \sqrt{1 + 4\epsilon^4} &= 1 + \frac{1}{2}(4\epsilon^4) - \frac{1}{8}(4\epsilon^4)^2 + O(\epsilon^{12}) && (129)\\ &= 1 + 2\epsilon^4 - 2\epsilon^8 + O(\epsilon^{12}). && (130) \end{aligned}$$

We divide the previous equation by ϵ and get

$$\sqrt{1/\epsilon^2 + 4\epsilon^2} = \frac{1}{\epsilon} + 2\epsilon^3 - 2\epsilon^7 + O(\epsilon^{11}). \tag{131}$$

This implies

$$\begin{aligned} -1/\epsilon - \sqrt{1/\epsilon^2 + 4\epsilon^2} &= -\frac{1}{\epsilon} - \frac{1}{\epsilon} - 2\epsilon^3 + 2\epsilon^7 + O(\epsilon^{11}), && (132)\\ &= -\frac{2}{\epsilon} - 2\epsilon^3 + O(\epsilon^7), && (133)\\ -1/\epsilon + \sqrt{1/\epsilon^2 + 4\epsilon^2} &= -\frac{1}{\epsilon} + \frac{1}{\epsilon} + 2\epsilon^3 - 2\epsilon^7 + O(\epsilon^{11}) && (134)\\ &= 2\epsilon^3 - 2\epsilon^7 + O(\epsilon^{11}). && (135) \end{aligned}$$

We notice that the term $\frac{1}{\epsilon}$ has been *canceled* in the previous calculation. This cancelation generates the rounding error which is the topic of the associated numerical experiment. We divide the previous equations by 2ϵ and finally get

$$\begin{aligned} x_- &= -\frac{1}{\epsilon^2} - \epsilon^2 + O(\epsilon^6), && (136)\\ x_+ &= \epsilon^2 - \epsilon^6 + O(\epsilon^{10}). && (137) \end{aligned}$$

The two previous equations directly imply the approximations 126, when we consider that ϵ is close to zero. □

Answer of Exercise 2.7 (*Roots of a quadratic equation #2*) Let us prove the approximations 10. The two real solutions of the quadratic equation 15 are

$$x_- = \frac{-1/\epsilon - \sqrt{1/\epsilon^2 - 4}}{2}, \qquad x_+ = \frac{-1/\epsilon + \sqrt{1/\epsilon^2 - 4}}{2}. \tag{138}$$

Let us prove that

$$x_- \approx -1/\epsilon, \qquad x_+ \approx -\epsilon. \tag{139}$$

Based on the Taylor expansion 47, we can simplify the expression $\sqrt{1/\epsilon^2 - 4}$. Indeed, we have

$$\begin{aligned} \sqrt{1/\epsilon^2 - 4} &= \sqrt{\frac{1 - 4\epsilon^2}{\epsilon^2}} && (140)\\ &= \frac{\sqrt{1 - 4\epsilon^2}}{\epsilon}. && (141) \end{aligned}$$

Therefore,

$$\sqrt{1-4\epsilon^2} = 1+\frac{1}{2}(-4\epsilon^2)-\frac{1}{8}(-4\epsilon^2)^2+O(\epsilon^6) \quad (142)$$

$$= 1-2\epsilon^2-\frac{1}{2}\epsilon^4+O(\epsilon^6). \quad (143)$$

By equation 141, the previous equation can be divided by ϵ, which leads to

$$\sqrt{1/\epsilon^2-4} = \frac{1}{\epsilon}-2\epsilon-\frac{1}{2}\epsilon^3+O(\epsilon^5). \quad (144)$$

We now compute the expressions which appear in the calculation of the roots x_- and x_+. The previous equation leads to

$$-\frac{1}{\epsilon}-\sqrt{1/\epsilon^2-4} = -\frac{1}{\epsilon}-\frac{1}{\epsilon}+2\epsilon+\frac{1}{2}\epsilon^3+O(\epsilon^5), \quad (145)$$

$$= -\frac{2}{\epsilon}+2\epsilon+O(\epsilon^3), \quad (146)$$

$$-\frac{1}{\epsilon}+\sqrt{1/\epsilon^2-4} = -\frac{1}{\epsilon}+\frac{1}{\epsilon}-2\epsilon-\frac{1}{2}\epsilon^3+O(\epsilon^5), \quad (147)$$

$$= -2\epsilon-\frac{1}{2}\epsilon^3+O(\epsilon^5). \quad (148)$$

We divide the two previous equations by 2 and finally get:

$$x_- = -\frac{1}{\epsilon}+\epsilon+O(\epsilon^3), \quad (149)$$

$$x_+ = -\epsilon-\frac{1}{4}\epsilon^3+O(\epsilon^5). \quad (150)$$

The previous equations imply the approximations 139, when we consider that ϵ is close to zero. □

3 Numerical derivatives

In this section, we analyze 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 naive algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naive algorithm with the `derivative` Scilab function. In the third part, we analyze why and how floating point numbers must be taken into account when we compute numerical derivatives.

3.1 Theory

The basic result is the Taylor formula with one variable [22]. Assume that $x \in \mathbb{R}$ is a given number and $h \in \mathbb{R}$ is a given step. Assume that $f : \mathbb{R} \to \mathbb{R}$ is a two times continuously differentiable function. Therefore,

$$f(x+h) = f(x)+hf'(x)+\frac{h^2}{2}f''(x)+\mathcal{O}(h^3). \quad (151)$$

We immediately get the forward difference which approximates the first derivate at order 1

$$f'(x) = \frac{f(x+h)-f(x)}{h} + \frac{h}{2}f''(x) + \mathcal{O}(h^2). \quad (152)$$

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

input : x, h
output: $f'(x)$
$f'(x) := (f(x+h) - f(x))/h$;
Algorithm 3: Naive algorithm to compute the numerical derivative of a function of one variable.

3.2 Experiments

The following Scilab function `myfprime` is a straightforward implementation of the previous algorithm.

```
function fp = myfprime(f,x,h)
  fp = (f(x+h) - f(x))/h;
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 function `myfunction` computes the square function.

```
function y = myfunction (x)
  y = x*x;
endfunction
```

The (naive) idea is that the computed relative error is small when the step h is small. Because *small* is not a priori clear, we take $h = 10^{-16}$ as a "good" candidate for a *small* double.

The `derivative` function allows to compute the Jacobian and the Hessian matrix of a given function. Moreover, we can use formulas of order 1, 2 or 4. The `derivative` function has been designed by Rainer von Seggern and Bruno Pinçon. The order 1 formula is the forward numerical derivative that we have already presented.

In the following script, we compare the computed relative error produced by our naive method with step $h = 10^{-16}$ and the `derivative` function with default optimal step. We compare the two methods for the point $x = 1$.

```
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);
```

The previous script produces the following output.

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

Our naive method seems to be inaccurate and has no significant decimal digit. The Scilab function, instead, has 9 significant digits.

Since our faith is based on the truth of the mathematical theory, some deeper experiments must be performed. We make the following numerical experiment: we take the initial step $h = 1.0$ and divide h by 10 at each step of a loop made of 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
```

The previous script 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, gets to a minimum and then increases. 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

In this section, we make reasonable assumptions for the expression of the total error and compute the optimal step of a forward difference formula. We extend our work to the centered two points formula.

3.3.1 Floating point implementation

The first source of error is obviously the truncation error $E_t(h) = h|f''(x)|/2$, due to the limited Taylor expansion.

The other source of error is generated by the roundoff errors in the function evaluation of the formula $(f(x+h) - f(x))/h$. Indeed, the floating point representation of the function value at point x is

$$fl(f(x)) = (1 + e(x))f(x), \tag{153}$$

where the relative error e depends on the the current point x. We assume here that the relative error e is bounded by the product of a constant $c > 0$ and the machine precision r. Furthermore, we assume here that the constant c is equal to one. We may consider other rounding errors sources, such as the error in the sum $x + h$, the difference $f(x+h) - f(x)$ or the division $(f(x+h) - f(x))/h$. But all these rounding errors can be neglected for they are not, in general, as large as the roundoff error generated by the function evaluation. Hence, the roundoff error associated with the function evaluation is $E_r(h) = r|f(x)|/h$.

Therefore, the total error associated with the forward finite difference is bounded by

$$E(h) = \frac{r|f(x)|}{h} + \frac{h}{2}|f''(x)|. \tag{154}$$

The error is then the sum of a term which is a decreasing function of h and a term which an increasing function of h. We consider the problem of finding the step h which minimizes the error $E(h)$. The total error $E(h)$ is minimized when its first derivative is zero. The first derivative of the function E is

$$E'(h) = -\frac{r|f(x)|}{h^2} + \frac{1}{2}|f''(x)|. \tag{155}$$

The second derivative of E is

$$E''(h) = 2\frac{r|f(x)|}{h^3}. \tag{156}$$

If we assume that $f(x) \neq 0$, then the second derivative $E''(h)$ is strictly positive, since $h > 0$ (i.e. we consider only non-zero steps). Hence, there is only one global solution of the minimization problem. This first derivative is zero if and only if

$$-\frac{r|f(x)|}{h^2} + \frac{1}{2}|f''(x)| = 0 \tag{157}$$

Therefore, the optimal step is

$$\overline{h} = \sqrt{\frac{2r|f(x)|}{|f''(x)|}}. \tag{158}$$

Let us make the additional assumption

$$\frac{2|f(x)|}{|f''(x)|} \approx 1. \tag{159}$$

Then the optimal step is

$$\overline{h} = \sqrt{r}, \tag{160}$$

where the error is

$$E(\overline{h}) = 2\sqrt{r}. \tag{161}$$

With double precision floating point numbers, we have $r = 10^{-16}$ and we get $\overline{h} = 10^{-8}$ and $E(\overline{h}) = 2.10^{-8}$. Under our assumptions on f and on the form of the total error, this is the minimum error which is achievable with a forward difference numerical derivate.

We can extend the previous method to the first derivate computed by a centered 2 points formula. We can prove that

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \frac{h^2}{6} f'''(x) + \mathcal{O}(h^3). \tag{162}$$

We can apply the same method as previously and, under reasonable assumptions on f and the form of the total error, we get that the optimal step is $h = r^{1/3}$, which corresponds to the total error $E = 2r^{2/3}$. With double precision floating point numbers, this corresponds to $h \approx 10^{-5}$ and $E \approx 10^{-10}$.

3.3.2 Robust algorithm

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

$h := \sqrt{r}$;
$f'(x) := (f(x+h) - f(x))/h$;

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

3.4 One more step

In this section, we analyze the behavior of the `derivative` function when the point x is either large in magnitude, small or close to zero. We compare these results with the `numdiff` function, which does not use the same step strategy. As we are going to see, both functions performs the same when x is near 1, but performs very differently when x is large or small.

The `derivative` function uses the optimal step based on the theory we have presented. But the optimal step does not solve all the problems that may occur in practice, as we are going to see.

See for example the following Scilab session, where we compute the numerical derivative of $f(x) = x^2$ for $x = 10^{-100}$. The expected result is $f'(x) = 2. \times 10^{-100}$.

```
-->fp = derivative(myfunction,1.e-100,order=1)
 fp  =
    0.000000014901161193847
```

```
-->fe=2.e-100
 fe  =
    2.000000000000000040-100
-->e = abs(fp-fe)/fe
 e  =
    7.450580596923828243D+91
```

The result does not have any significant digit.

The explanation is that the step is $h = \sqrt{r} \approx 10^{-8}$. Then, the point $x + h$ is computed as $10^{-100} + 10^{-8}$ which is represented by a floating point number which is close to 10^{-8}, because the term 10^{-100} is much smaller than 10^{-8}. Then we evaluate the function, which leads to $f(x+h) = f(10^{-8}) = 10^{-16}$. The result of the computation is therefore $(f(x + h) - f(x))/h = (10^{-16} + 10^{-200})/10^{-8} \approx 10^{-8}$.

That experiment shows that the `derivative` function uses a poor default step h when x is very small.

To improve the accuracy of the computation, we can take the control of the step h. A reasonable solution is to use $h = \sqrt{r}|x|$ 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  =
    0.000000006549569677079 4
```

But when x is exactly zero, the step $h = \sqrt{r}|x|$ cannot work, because it would produce the step $h = 0$, which would generate a division by zero exception. In that case, the step $h = \sqrt{r}$ provides a sufficiently good accuracy.

Another function is available in Scilab to compute the numerical derivatives of a given function, that is `numdiff`. The `numdiff` function uses the step

$$h = \sqrt{r}(1 + 10^{-3}|x|). \tag{163}$$

In the following paragraphs, we analyze why this formula has been chosen. As we are going to check experimentally, this step formula performs better than `derivative` when x is large, but performs equally bad when x is small.

As we can see the following session, the behavior 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 the `derivative` function. But the number of significant digits is approximately the same, i.e. 9 digits.

The goal of the step used by the `numdiff` function is to produce good accuracy when the value of x is large. In this case, the `numdiff` function produces accurate results, while the `derivative` function performs poorly.

In the following session, we compute the numerical derivative of the function $f(x) = x^2$ at the point $x = 10^{10}$. The expected result is $f'(x) = 2.10^{10}$.

```
-->numdiff(myfunction,1.e10)
 ans  =
   2.000D+10
-->derivative(myfunction,1.e10,order=1)
 ans  =
   0.
```

We see that the `numdiff` function produces an accurate result while the `derivative` function produces a result which has no significant digit.

The behavior of the two functions when x is close to zero is the same, i.e. both functions produce wrong results. Indeed, when we use the `derivative` function, the step $h = \sqrt{r}$ is too large so that the point x is neglected against the step h. On the other hand, we we use the `numdiff` function, the step $h = \sqrt{r}(1 + 10^{-3}|x|)$ is approximated by $h = \sqrt{r}$ so that it produces the same results as the `derivative` function.

3.5 References

A reference for numerical derivatives is [6], chapter 25. "Numerical Interpolation, Differentiation and Integration" (p. 875). The webpage [43] and the book [39] give results about the rounding errors.

In order to solve this issue generated by the magnitude of x, more complex methods should be used. Moreover, we did not give the solution of other sources of rounding errors. Indeed, the step $h = \sqrt{r}$ was computed based on assumptions on the rounding error of the function evaluations, where we consider that the constant c is equal to one. This assumption is satisfied only in the ideal case. Furthermore, we make the assumption that the factor $\frac{2|f(x)|}{|f''(x)|}$ is close to one. This assumption is far from being achieved in practical situations, where the function value and its second derivative can vary greatly in magnitude.

Several authors attempted to solve the problems associated with numerical derivatives. A non-exhaustive list of references includes [28, 11, 45, 16].

4 Complex division

In this section, we analyze the problem of the complex division in Scilab. We especially detail the difference between the mathematical, straightforward formula and the floating point implementation. In the first part, we briefly report the formulas which allow to compute the real and imaginary parts of the division of two complex

numbers. We then present the naive algorithm based on these mathematical formulas. In the second part, we make some experiments in Scilab and compare our naive algorithm with Scilab's division operator. In the third part, we analyze why and how floating point numbers must be taken into account when the implementation of such division is required.

4.1 Theory

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

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}. \tag{164}$$

So that the real and imaginary parts e and f of the complex number are

$$e = \frac{ac+bd}{c^2+d^2}, \tag{165}$$

$$f = \frac{bc-ad}{c^2+d^2}. \tag{166}$$

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

input : a, b, c, d
output: e, f
$den := c^2 + d^2$;
$e := (ac + bd)/den$;
$f := (bc - ad)/den$;

Algorithm 5: Naive algorithm to compute the complex division. The algorithm takes as input the real and imaginary parts a, b, c, d of the two complex numbers and returns e and f, the real and imaginary parts of the division.

4.2 Experiments

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

$$\frac{1+i2}{3+i4} = \frac{11}{25} + i\frac{2}{25} = 0.44 + i0.08. \tag{167}$$

We check our result with Wolfram Alpha[42], with the input "(1+i*2)/(3+i*4)". In the following script, we check that there is no obvious bug in the naive implementation.

```
-->  [e f] = naive ( 1.0 , 2.0 , 3.0 , 4.0 )
 f  =
    0.08
 e  =
    0.44
-->  (1.0 + %i * 2.0 )/(3.0 + %i * 4.0 )
 ans  =
    0.44 + 0.08i
```

The results of the `naive` function and the division operator are the same, which makes us confident that our implementation is correct.

Now that we are confident, we make the following numerical experiment involving a large number. Consider the complex division

$$\frac{1+i}{1+i10^{307}} \approx 1.0000000000000000 \cdot 10^{-307} - i1.0000000000000000 \cdot 10^{-307}, \tag{168}$$

which is accurate to the displayed digits. We check our result with Wolfram Alpha[42], with the input "(1 + i)/(1 + i * 10^307)". 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

$$\frac{1+i}{10^{-307}+i10^{-307}} = \frac{1+i}{10^{-307}(1+i)} = 10^{307}. \tag{169}$$

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.

4.3 Explanations

In this section, we analyze the reason why the naive implementation of the complex division leads to inaccurate results. In the first section, we perform algebraic computations and shows the problems of the naive formulas. In the second section, we present the Smith's method.

4.3.1 Algebraic computations

In this section, we analyze the results produced by the second and third tests in the previous numerical experiments. We show that the intermediate numbers which appear are not representable as double precision floating point numbers.

Let us analyze the second complex division 168. We are going to see that this division is associated with an IEEE overflow. We have $a = 1$, $b = 1$, $c = 1$ and $d = 10^{307}$. By the equations 165 and 166, we have

$$\begin{aligned} den &= c^2 + d^2 = 1^2 + (10^{307})^2 && (170)\\ &= 1 + 10^{614} \approx 10^{614}, && (171)\\ e &= (ac + bd)/den = (1 * 1 + 1 * 10^{307})/10^{614}, && (172)\\ &\approx 10^{307}/10^{614} \approx 10^{-307}, && (173)\\ f &= (bc - ad)/den = (1 * 1 - 1 * 10^{307})/10^{614} && (174)\\ &\approx -10^{307}/10^{614} \approx -10^{-307}. && (175) \end{aligned}$$

We see that both the input numbers a, b, c, d are representable and the output numbers $e = 10^{-307}$ and $f = -10^{-307}$ are representable as double precision floating point numbers. We now focus on the floating point representation of the intermediate expressions. We have

$$fl(den) = fl(10^{614}) = Inf, \quad (176)$$

because 10^{614} is not representable as a double precision number. Indeed, the largest positive double is 10^{308}. The IEEE Inf floating point number stands for Infinity and is associated with an overflow. The Inf floating point number is associated with an algebra which defines that $1/Inf = 0$. This is consistent with mathematical limit of the function $1/x$ when $x \to \infty$. Then, the e and f terms are computed as

$$\begin{aligned} fl(e) &= fl((ac + bd)/den) = fl((1 * 1 + 1 * 10^{307})/Inf) = fl(10^{307}/Inf) = 0, && (177)\\ fl(f) &= fl((bc - ad)/den) = fl((1 * 1 - 1 * 10^{307})/Inf) = fl(-10^{307}/Inf) = -0. && (178) \end{aligned}$$

Hence, the result is computed without any significant digit, even though both the input and the output numbers are all representable as double precision floating point numbers.

Let us analyze the second complex division 169. We are going to see that this division is associated with an IEEE underflow. We have $a = 1$, $b = 1$, $c = 10^{-307}$ and $d = 10^{-307}$. We now use the equations 165 and 166, which leads to:

$$\begin{aligned} den &= c^2 + d^2 = (10^{-307})^2 + (10^{-307})^2 && (179)\\ &= 10^{-614} + 10^{-614} = 2.10^{-616}, && (180)\\ e &= (ac + bd)/den = (1 * 10^{-307} + 1 * 10^{-307})/(2.10^{-614}) && (181)\\ &= (2.10^{-307})/(2.10^{-614}) = 10^{307}, && (182)\\ f &= (bc - ad)/den = (1 * 10^{-307} - 1 * 10^{-307})/(2.10^{-614}) && (183)\\ &= 0/10^{-614} = 0. && (184) \end{aligned}$$

With double precision floating point numbers, the computation is not performed this way. The positive terms which are smaller than 10^{-324} are too small to be representable in double precision and are represented by 0 so that an underflow occurs. This leads to

$$\begin{aligned} fl(den) &= fl(c^2 + d^2) = fl(10^{-614} + 10^{-614}) && (185)\\ &= 0, && (186)\\ fl(e) &= fl((ac + bd)/den) = fl((1 * 10^{-307} + 1 * 10^{-307})/(2.10^{-614})) && (187)\\ &= fl(2.10^{-307}/0) = Inf, && (188)\\ fl(f) &= fl((bc - ad)/den) = fl((1 * 10^{-307} - 1 * 10^{-307})/0) && (189)\\ &= fl(0/0) = NaN. && (190) \end{aligned}$$

The two previous examples shows that, even if both the input and output numbers are representable as floating point numbers, the intermediate expressions may generate numbers which may not be representable as floating point numbers. Hence, a naive implementation can lead to inaccurate results. In the next section, we present a method which allows to cure most problems generated by the complex division.

4.3.2 Smith's method

In this section, we analyze Smith's method, which allows to produce an accurate division of two complex numbers. We present the detailed steps of this modified algorithm in the particular cases that we have presented.

In Scilab, the algorithm which allows to perform the complex division is done by the the *wwdiv* routine, which implements Smith's method [44]. This implementation is due to Bruno Pinçon. 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 164, which is reproduced here for simplicity

$$\frac{a + ib}{c + id} = e + if = \frac{ac + bd}{c^2 + d^2} + i\frac{bc - ad}{c^2 + d^2}. \quad (191)$$

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.

Consider the term $e = \frac{ac+bd}{c^2+d^2}$ in the equation 191 and assume that $c \neq 0$ and $d \neq 0$. Let us assume that c is large in magnitude with respect to d, i.e. $|d| \ll |c|$. This implies $d/c \leq 1$. We see that the denominator $c^2 + d^2$ squares the number c, which also appears in the numerator. Therefore, we multiply both the numerator and the denominator by $1/c$. If we express e as $\frac{a+b(d/c)}{c+d(d/c)}$, which is mathematically equivalent, we see that there is no more squaring of c. Hence, overflows are less likely to occur in the denominator, since $|d(d/c)| = |d||d/c| \leq |d|$. That is, there is no growth in the magnitude of the terms involved in the computation of the product $d(d/c)$. Similarly, overflows are less likely to occur in the numerator, since $|b(d/c)| = |b||d/c| \leq |b|$. In the opposite case where d is large with respect to c, i.e. $d \gg c$, we could divide the numerator and the denominator by $1/d$. This leads to the formulas

$$\frac{a+ib}{c+id} = \frac{a+b(d/c)}{c+d(d/c)} + i\frac{b-a(d/c)}{c+d(d/c)}, \tag{192}$$

$$= \frac{a(c/d)+b}{c(c/d)+d} + i\frac{b(c/d)-a}{c(c/d)+d}. \tag{193}$$

The previous equations can be simplified as

$$\frac{a+ib}{c+id} = \frac{a+br}{c+dr} + i\frac{b-ar}{c+dr}, \qquad r = d/c, \text{ if } |c| \geq |d|, \tag{194}$$

$$= \frac{ar+b}{cr+d} + i\frac{br-a}{cr+d}, \qquad r = c/d, \text{ if } |d| \geq |c|. \tag{195}$$

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
```

We now check that Smith's method performs very well for the difficult complex division that we met earlier in this chapter.

Let us analyze the second complex division 168. We have $a = 1$, $b = 1$, $c = 1$ and $d = 10^{307}$. For this division, Smith's method is the following.

```
if ( |1.e307| <= |1| ) > test false
else
  r = c/d = 1 / 1.e307 = 1.e-307
  den  = c * r + d = 1 * 1.e-307 + 1.e307 = 1.e307
  e = (a * r + b)/den = (1 * 1.e-307 + 1) / 1.e307 = 1 / 1.e307
    = 1.e-307
  f = (b * r - a)/den = (1 * 1.e-307 - 1) / 1.e307 = -1 / 1.e307
    = -1.e-308
```

We see that, while the naive division generated an overflow, Smith's method produces the correct result.

Let us analyze the second complex division 169. We have $a = 1$, $b = 1$, $c = 10^{-307}$ and $d = 10^{-307}$.

```
if ( |1.e-307| <= |1.e-307| ) > test true
  r = d/c = 1.e-307 / 1.e-307 = 1
  den  = c + d * r = 1.e-307 +  1e-307 * 1 = 2.e-307
  e = (a + b * r) / den = (1 + 1 * 1) / 2.e-307 = 2/2.e-307
    = 1.e307
  f = (b - a * r) / den = (1 - 1 * 1) / 2.e-307
    = 0
```

We see that, while the naive division generated an underflow, Smith's method produces the correct result.

Now that we have designed a more robust algorithm, we are interested in testing Smith's method on a more difficult case.

4.4 One more step

In this section, we show the limitations of Smith's method and present an example where Smith's method does not perform as expected.

The following example is inspired by an example by Stewart's in [46]. 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

$$\frac{10^{307} + i10^{-307}}{10^{204} + i10^{-204}} \approx 1.000000000000000 \cdot 10^{103} - i1.000000000000000 \cdot 10^{-305}, \quad (196)$$

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 first check why the naive implementation is not accurate in this case. We have $a = 10^{307}$, $b = 10^{-307}$, $c = 10^{204}$ and $d = 10^{-204}$. Indeed, the naive implementation performs the following steps.

```
den = c * c + d * d = 1.e204 * 1.e204 + 1.e-204 * 1.e-204
    = Inf
e = (a * c + b * d) / den
  = (1.e307 * 1.e204 + 1.e-307 * 1.e-204 ) / Inf = Inf / Inf
  = Nan
f = (b * c - a * d) / den
  = (1.e-307 * 1.e204 - 1.e307 * 1.e-204) / Inf = -1.e103 / Inf
  = 0
```

We see that the denominator `den` overflows, which makes `e` to be computed as `Nan` and `f` to be computed as `0`.

Second, 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 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.

Finally, we 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 [37, 34]. The processor being used may offer an internal precision that is wider than the precision

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.

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 [9], 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 [13], 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.

4.5 References

The 1962 paper by R. Smith [44] describes the algorithm which is used in Scilab.

Goldberg introduces in [18] many of the subjects presented in this document,

including the problem of the complex division.

An analysis of Hough, cited by Coonen [8] and Stewart [46] shows that when the algorithm works, it returns a computed value $\overline{z}$ satisfying

$$|\overline{z} - z| \leq \epsilon |z|, \tag{197}$$

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 [46]. 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 $x_1 \ldots x_n$ 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.

Stewart's algorithm is superseded by the one by Li et al. [31], but also by Kahan's [25], which, from [40], is the one implemented in the C99 standard.

In [31], 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. Notice that this appendix is presented in the technical report (October 20, 2000), but not in the paper published by ACM (2002).

In the ISO/IEC 9899:TC3 C Committee Draft [21], 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 [27] (in the Appendix "Over/Underflow Undermines Complex Number Division in Java"), this code is due to Jim Thomas and Fred Tydeman.

Knuth presents in [29] the Smith's method in section 4.2.1, as exercize 16. Knuth gives also references [48] and [15]. The 1967 paper by Friedland [15] describes two algorithms to compute the absolute value of a complex number $|x + iy| = \sqrt{x^2 + y^2}$ and the square root of a complex number $\sqrt{x + iy}$.

Issues related to the use of extended double precision floating point numbers are analyzed by Muller et al. in [37]. In the section 3 of part I, "Floating point formats an Environment", the authors analyze the "double rounding" problem which occurs when an internal precision is wider than the precision of the variables of a program. The typical example is the double-extended format available on Intel platforms. Muller et al. show different examples, where the result depends on the compiler options and the platform, including an example extracted from a paper by Monniaux [34].

Corden and Kreitzer analyse in [9] the effect of the Intel compiler floating point options on the numerical results. The paper focuses on the reproductibility issues which are associated with floating point computations. The options which allow to be compliant with the IEEE standards for C++ and Fortran are presented. The effects of optimization options is considered with respect to speed and the safety of the transformations that may be done on the source code.

The "Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture" [10] is part of a set of documents that describes the architecture and programming environment of Intel 64 and IA-32 architecture processors. The chapter 8, "Programming with the x87 environment" presents the registers and the instruction set for this unit. The section 8.1.2, "x87 FPU Data Registers" focuses on the floating point registers, which are based on 80-bits and implements the double extended-precision floating-point format. The chapter 10, "Programming with Streaming SIMD Extensions (SSE)" introduces the extensions which were introduced into the IA-32 architecture in the Pentium III processor family. The chapter 11 introduces the SSE2 extensions.

In [34], David Monniaux presents issues related to the analysis of floating point programs. He emphasizes the difficulty of defining the semantics of common implementation of floating point numbers, depending on choices made by the compiler. He gives concrete examples of problems that can appear and solutions.

In the exercise 25.2 of the chapter 25 "Software issues in floating point arithmetic" of [20], 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 $(c+id)(e+if) = a + ib$.

In the Appendix B "Smith?s Complex Division Algorithm with Scaling", of [30], Li et al. present a modified Smith's algorithm.

4.6 Exercises

Exercise 4.1 (*Complex division formula*) Prove equality 164.

Exercise 4.2 (*Complex division*) Prove that $\frac{1+i2}{3+i4} = \frac{11}{25} + i\frac{2}{25}$.

Exercise 4.3 (*Complex division*) Prove a simplified version of the equation 168, that is, prove that

$$\frac{1+i}{1+i10^{307}} \approx 10^{-307} - i10^{-307}. \tag{198}$$

Exercise 4.4 (*Taylor expansion of inverse function near zero*) Prove that if $x \in \mathbb{R}$ is close to zero, then

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + O(x^4). \tag{199}$$

Exercise 4.5 (*Taylor expansion of inverse function near* $+\infty$) Prove that if $x \in \mathbb{R}$ is close to $+\infty$, then

$$\frac{1}{1+x} = \frac{1}{x} - \left(\frac{1}{x}\right)^2 + \left(\frac{1}{x}\right)^3 + O\left(\left(\frac{1}{x}\right)^4\right). \tag{200}$$

Exercise 4.6 (*Complex division approximation*) Prove that the approximation 168 is accurate to more that 300 digits, that is, prove that

$$\frac{1+10^{307}}{1+10^{614}} = 10^{-307} + O(10^{-614}). \tag{201}$$

Exercise 4.7 (*Complex division approximation*) Assume that m, n are integers which satisfy

$$m \gg 0 \tag{202}$$
$$n \gg 0 \tag{203}$$
$$n \gg m \tag{204}$$

Prove that

$$\frac{10^n + i10^{-n}}{10^m + i10^{-m}} \approx 10^{n-m} - i10^{n-3m}. \tag{205}$$

Exercise 4.8 (*Complex division approximation*) Assume that the integers m and n satisfy

$$4 < m \leq 308, \tag{206}$$
$$12 < n \leq 308, \tag{207}$$
$$m + 8 < n, \tag{208}$$
$$0 \leq n - m \leq 308, \tag{209}$$
$$-307 \leq n - 3m \leq 0. \tag{210}$$

Prove that the approximation 205 is an equality with double precision floating point numbers and prove that all the terms in the expression 205 are representable as floating point numbers.

Exercise 4.9 (*Examples of failing Smith's formula*) Assume that the integers m and n satisfy the inequalities 206 to 210. Assume that the integers m and n satisfy the inequalities

$$162 < m, \tag{211}$$
$$324 < m + n. \tag{212}$$

Prove that Smith's method fails to compute a correct result for the complex division 205 with double floating point numbers, that is, prove that the imaginary part computed by Smith's method is zero.

Exercise 4.10 (*Examples of failing Smith's formula*) Compute the number of integers m and n satisfying the assumptions of exercise 4.8. Give examples of such integers m and n.

Compute the number of integers m and n satisfying the assumptions of exercise 4.9. Give examples of such integers m and n.

4.7 Answers to exercises

Answer of Exercise 4.1 (*Complex division formula*) Let us prove equality 164. We consider the fraction $\frac{a+ib}{c+id}$ and we multiply both the numerator and the denominator by the number $\overline{c+id} = c - id$. This leads to

$$\frac{a+ib}{c+id} = \frac{a+ib}{c+id}\frac{c-id}{c-id} \tag{213}$$
$$= \frac{(ac - i^2bd) + i(bc - ad)}{c^2 + d^2} \tag{214}$$
$$= \frac{(ac + bd) + i(bc - ad)}{c^2 + d^2}. \tag{215}$$

We separate the real and imaginary parts in the previous equations and finally get the equality 164. □

Answer of Exercise 4.2 (*Complex division formula*) By the equation 164, we have

$$\begin{aligned}\frac{1+i2}{3+i4} &= \frac{1\cdot 3+2\cdot 4}{3^2+4^2}+i\frac{2\cdot 3-1\cdot 4}{3^2+4^2} && (216)\\ &= \frac{3+8}{9+16}+i\frac{6-4}{9+16} && (217)\\ &= \frac{11}{25}+i\frac{2}{25}. && (218)\end{aligned}$$

□

Answer of Exercise 4.3 (*Complex division*) By the equation 164, we have

$$\begin{aligned}\frac{1+i}{1+10^{307}i} &= \frac{1\cdot 1+1\cdot 10^{307}}{1^2+(10^{307})^2}+i\frac{1\cdot 1-1\cdot 10^{307}}{1^2+(10^{307})^2} && (219)\\ &= \frac{1+10^{307}}{1+10^{614}}+i\frac{1-10^{307}}{1+10^{614}} && (220)\\ &\approx \frac{10^{307}}{10^{614}}-i\frac{10^{307}}{10^{614}} && (221)\\ &\approx 10^{-307}-i10^{-307}. && (222)\end{aligned}$$

□

Answer of Exercise 4.4 (*Taylor expansion of inverse function near zero*) Assume that f is a continuously differentiable function. By Taylor's theorem, we have

$$f(x+h)=f(x)+hf'(x)+\frac{1}{2}h^2f''(x)+\frac{1}{6}h^3f'''(x)+O(h^4). \quad (223)$$

We use the Taylor's expansion 223 with $f(x)=\frac{1}{1+x}$ in the neighborhood of $x=0$. The derivatives of the function f are

$$f'(x)=-(1+x)^{-2}, \qquad f''(x)=2(1+x)^{-3}, \qquad f'''(x)=-6(1+x)^{-4}, \quad (224)$$

so that

$$f(0)=1, \qquad f'(0)=-1, \qquad f''(0)=2, \qquad f'''(0) = -6. \quad (225)$$

The Taylor expansion 223 therefore implies

$$\begin{aligned}\frac{1}{1+h} &= 1+h\cdot(-1)+\frac{1}{2}h^2\cdot 2+\frac{1}{6}h^3\cdot(-6)+O(h^4), && (226)\\ &= 1-h+h^2-h^3+O(h^4), && (227)\end{aligned}$$

which concludes the proof. □

Answer of Exercise 4.5 (*Taylor expansion of inverse function near $+\infty$*) When $x\to+\infty$, we have $1/x\to 0$. Therefore, we can use the result of the exercise 4.4, which implies

$$\frac{1}{1+1/x} = 1-(1/x)+(1/x)^2-(1/x)^3+O((1/x)^4). \quad (228)$$

We can simplify the previous expression. Indeed,

$$\begin{aligned}\frac{1}{1+1/x} &= \frac{1}{\frac{1+x}{x}} && (229)\\ &= \frac{x}{1+x} && (230)\\ &= \frac{x+1-1}{1+x} && (231)\\ &= 1-\frac{1}{1+x}. && (232)\end{aligned}$$

By the equation 228, this leads to

$$1 - \frac{1}{1+x} = 1 - (1/x) + (1/x)^2 - (1/x)^3 + O((1/x)^4). \tag{233}$$

In the previous equation, the constant term 1 can be simplified from both sides, which immediately leads to the equation 200 and concludes the proof. □

Answer of Exercise 4.6 (*Complex division approximation*) Let us prove that the approximation 168 is accurate to more that 300 digits, that is, let us prove that

$$\frac{1+10^{307}}{1+10^{614}} = 10^{-307} + O(10^{-614}). \tag{234}$$

The starting point is the equality 220, which we rewrite here for consistency:

$$\frac{1+i}{1+i10^{307}} = \frac{1+10^{307}}{1+10^{614}} + i\frac{1-10^{307}}{1+10^{614}}. \tag{235}$$

We use the Taylor expansion 200 in order to compute the expression $\frac{1}{1+10^{614}}$, and then compute the fraction $\frac{1+10^{307}}{1+10^{614}}$. By the Taylor expansion 200, we have

$$\frac{1}{1+10^{614}} = \frac{1}{10^{614}} + O\left(\left(\frac{1}{10^{614}}\right)^2\right) \tag{236}$$

$$= 10^{-614} + O(10^{-1228}). \tag{237}$$

Therefore,

$$\frac{1+10^{307}}{1+10^{614}} = (1+10^{307})(10^{-614} + O(10^{-1228})) \tag{238}$$

$$= 10^{-614} + O(10^{-1228}) + 10^{-307} + O(10^{-921}) \tag{239}$$

$$= 10^{-307} + O(10^{-614}). \tag{240}$$

The last equality proves that the approximation $\frac{1+10^{307}}{1+10^{614}} \approx 1.0\ldots0 \cdot 10^{-307}$ is accurate up to 306 zeros. Similarily, we have

$$\frac{1-10^{307}}{1+10^{614}} = (1-10^{307})(10^{-614} + O(10^{-1228})) \tag{241}$$

$$= 10^{-614} + O(10^{-1228}) - 10^{-307} + O(10^{-921}) \tag{242}$$

$$= -10^{-307} + O(10^{-614}). \tag{243}$$

Therefore, the approximation $\frac{1-10^{307}}{1+10^{614}} \approx -1.0\ldots0 \cdot 10^{-307}$ is accurate up to 306 zeros. This proves that the approximation 168 is accurate to more that 300 digits and concludes the proof. □

Answer of Exercise 4.7 (*Complex division approximation*) By the equation 164, we have

$$\frac{10^n + i10^{-n}}{10^m + i10^{-m}} = \frac{10^n 10^m + 10^{-n}10^{-m}}{(10^m)^2 + (10^{-m})^2} + i\frac{10^{-n}10^m - 10^n 10^{-m}}{(10^m)^2 + (10^{-m})^2} \tag{244}$$

$$= \frac{10^{n+m} + 10^{-(m+n)}}{10^{2m} + 10^{-2m}} + i\frac{10^{m-n} - 10^{n-m}}{10^{2m} + 10^{-2m}}. \tag{245}$$

We will now use the assumptions 202 to 204 and simplify the previous equation.

- The assumptions 202 and 203 implies $m + n \gg 0$. Therefore,

$$10^{n+m} + 10^{-(m+n)} \approx 10^{n+m}. \tag{246}$$

- By the assumption 202, we have

$$10^{2m} + 10^{-2m} \approx 10^{2m}. \tag{247}$$

- The assumption 204 implies $n - m \gg 0$. Therefore,

$$10^{m-n} - 10^{n-m} \approx -10^{n-m}. \tag{248}$$

Therefore, the equation 245 simplifies into

$$\begin{aligned} \frac{10^n + i10^{-n}}{10^m + i10^{-m}} &\approx \frac{10^{n+m}}{10^{2m}} + i\frac{-10^{n-m}}{10^{2m}} && (249) \\ &\approx 10^{n-m} - i10^{n-3m}. && (250) \end{aligned}$$

We could prove that this approximation is correct up to approximately several 100 digits. This would require to use a Taylor expansion, as we did previously. □

Answer of Exercise 4.8 (*Complex division approximation*) Assume that we consider IEEE double precision floating point numbers. Assume that x and y are integers and satisfy $-307 \leq x, y \leq 308$. Double precision numbers are associated with at most 16 significant decimal digits. Therefore, if $x > y + 16$, we have

$$fl(10^x + 10^y) = fl(10^x). \tag{251}$$

We will use this property throughout this exercise.

We are going to prove that, under the stated assumptions, we have

$$fl\left(\frac{10^n + i10^{-n}}{10^m + i10^{-m}}\right) = fl(10^{n-m} - i10^{n-3m}). \tag{252}$$

We consider the equality 245 and consider the particular range of integers which are so that the approximations that we derived from there are satisfied with doubles.

- By the equality 251, the equality

$$fl(10^{n+m} + 10^{-(m+n)}) = fl(10^{n+m}) \tag{253}$$

  is true if $n + m > -(m + n) + 16$. This is equivalent to $2(m + n) > 16$ which can be written

$$m + n > 8. \tag{254}$$

- By the equality 251, the equality

$$fl(10^{2m} + 10^{-2m}) = fl(10^{2m}) \tag{255}$$

  is true if $2m > -2m + 16$. This is equivalent to $4m > 16$ which can be written

$$m > 4. \tag{256}$$

- By the equality 251, the equality

$$fl(10^{m-n} - 10^{n-m}) = fl(-10^{n-m}) \tag{257}$$

  is true if $n - m > m - n + 16$. This is equivalent to $2(n - m) > 16$ and can be written as

$$n > m + 8. \tag{258}$$

Notice that, if $m > 4$ and $n > m + 8$, then $n > 4 + 8 = 12$ which is implied by the inequality 207. If the inequalities 254, 256 and 258 are satisfied, then the equality 252 is satisfied. Assume that the inequalities 206, 207 and 208 are satisfied. Let us prove that this imply that the inequalities 254, 256 and 258 are satisfied.

- By the inequalities 206 and 207, we have $m > 4$ and $n > 12$, which imply $m + n > 16$ so that the inequality 254 is satisfied.
- By the inequality 206, we have $m > 4$, so that 256 is satisfied.

- By assumption, we have 208, which is identical to 258.

Therefore, we have prove that the inequalities 206, 207 and 208 leads the floating point equality 252.

It is easy to prove that when the inequalities 206 to 210 are satisfied, then all the terms in the complex division 252 are representable as double precision floating point numbers. □

Answer of Exercise 4.9 (*Examples of failing Smith's formula*) Assume that the integers m and n satisfy the inequalities 206 to 210. Let us consider the steps of Smith's method in the particular case of the complex division 252. We have $a = 10^n$, $b = 10^{-n}$, $c = 10^m$ and $d = 10^{-m}$. Since $|d| = 10^{-m} \leq |c| = 10^m$, the floating point statements of Smith's method are the following.

$$
\begin{aligned}
fl(r) &= fl(d/c) = fl(10^{-m}/10^m) = fl(10^{-2m}) && (259)\\
fl(den) &= fl(c + dr) = fl(10^m + 10^{-m}10^{-2m}) = fl(10^m + 10^{-3m}) && (260)\\
&= fl(10^m) && (261)\\
fl(e) &= fl((a + br)/den) = fl((10^n + 10^{-n}10^{-2m})/(10^m + 10^{-3m})) && (262)\\
&= fl((10^n + 10^{-n-2m})/(10^m + 10^{-3m})) = fl(10^n/10^m) && (263)\\
&= fl(10^{n-m}) && (264)\\
fl(f) &= fl((b - ar)/den) = fl((10^{-n} - 10^n10^{-2m})/(10^m + 10^{-3m})) && (265)\\
&= fl((10^{-n} - 10^{n-2m})/(10^m + 10^{-3m})) = fl(-10^{n-2m}/10^m) && (266)\\
&= fl(-10^{n-3m}) && (267)
\end{aligned}
$$

We now justify the approximations that we have used in the previous computation, which are mainly based on the equality 251.

- We have $fl(10^m + 10^{-3m}) = fl(10^m)$. Indeed, by assumption 206, we have $m > 4$, which implies $4m > 16$. This leads to $m > -3m + 16$ and the equation 251 proves the result.
- We have $fl(10^n + 10^{-n-2m}) = fl(10^n)$. Indeed, by the assumptions 206 and 207, we have $m > 4$ and $n > 12$, so that $m + n > 16 > 8$. Therefore, we have $2m + 2n > 16$, which implies $n > -n - 2m + 16$ and the equation 251 proves the result.
- We have $fl(10^{-n} - 10^{n-2m}) = fl(-10^{n-2m})$. Indeed, by the assumption 208, we have $n > m + 8$. This implies $n - m > 8$, which leads to $2n - 2m > 16$. We finally get $n - 2m > -n + 16$ and the equation 251 proves the result.

We now search conditions which produce a failure of Smith's method, that is, we prove the inequalities 211 and 212 leads to a failure of the previous formulas.

We notice that, if m is sufficiently large, then $r = 10^{-2m}$ cannot be represented as a double. This happens if $-2m < -324$, so that $r = 10^{-2m} < 10^{-324}$, which is the smallest positive nonzero double. The assumption $m > 162$ is implied by the inequality 211, which leads to $fl(r) = 0$. This implies that $fl(den) = fl(c + dr) = fl(c) = fl(10^m)$. This implies that $fl(f) = fl((b - ar)/den) = fl(b/den) = fl(10^{-n}/10^m) = fl(10^{-n-m})$. The imaginary part of the complex division f is zero if $-n - m < 324$. This is the assumption of the inequality 212. We have proved that the inequalities 211 and 212 leads to a zero floating point representation of f, the imaginary part of the complex division. □

Answer of Exercise 4.10 (*Examples of failing Smith's formula*) Let us compute the number of integers m and n satisfying the assumptions of exercise 4.8. The following Scilab script allows to compute these couples (n, m), by performing two nested loops in the allowed range for n and m.

```
N = 0;
for m = 5:308
  for n = 13:308
    if ( (m + 8 < n) & (n-m>=0) & (n-m <308) & ...
      (n-3*m >= -307) & (n-3*m<=0) ) then
      N = N + 1;
    end
```

```
    end
end
```

We find that there are 22484 such couples. In order to extract some examples from all these couples, we add the condition `modulo(n,20)==0 & modulo(m,20)==0` in the `if` statement. The following is a list of some examples.

```
n=40, m=20
n=220, m=80
n=140, m=120
n=260, m=140
n=280, m=160
```

It is easy to check our result with Wolfram Alpha[42], for example with the input "(10^140 + 10^-140 * i)/(10^20 + 10^-20 * i)".

Let us compute the number of integers m and n satisfying the assumptions of exercise 4.9. The following Scilab script allows to compute these couples (n, m), by performing two nested loops in the allowed range for n and m.

```
N = 0;
for m = 163:308
  for n = 13:308
    if ( (m + 8 < n) & (n-m>=0) & (n-m <308) ...
      & (n-3*m >= -307) & (n-3*m<=0) & (n+m>324) ) then
      N = N + 1;
    end
  end
end
```

We find 2752 such couples. In order to extract some examples from all these couples, we add the condition `modulo(n,20)==0 & modulo(m,20)==0` in the `if` statement. The following is a list of some examples.

```
n=240, m=180
n=260, m=180
n=280, m=180
n=300, m=180
n=300, m=200
```

We notice that 2752/22484= 0.122 (which is accurate to the displayed digits) that, when the complex division 252 is considered, it is not rare that Smith's method fails.

□

5 Conclusion

We have presented several cases where the mathematically perfect algorithm (i.e. without obvious bugs) does not produce accurate results with the computer in particular situations. Many Scilab algorithms take floating point values as inputs, and return floating point values as output. We have presented situations where the intermediate calculations involve terms which are not representable as floating point values. We have also presented examples where cancellation occurs so that the rounding errors dominate the result. We have analyzed specific algorithms which can be used to cure some of the problems.

Most algorithms provided by Scilab are designed specifically to take into account for floating point numbers issues. The result is a collection of robust algorithms which, most of the time, exceed the user's needs.

Still, it may happen that the algorithm used by Scilab is not accurate enough, so that floating point issues may occur in particular cases. We cannot pretend that Scilab always use the best algorithm. In fact, we have given in this document practical (but extreme) examples where the algorithm used by Scilab is not accurate. In this situation, an interesting point is that Scilab is open-source, so that anyone who wants to can inspect the source code, analyze the algorithm and point out the problems of this algorithm.

That article does not aim at discouraging from using floating point numbers or implementing our own algorithms. Instead, the goal of this document is to give examples where some specific work is to do when we translate the mathematical material into a computational algorithm based on floating point numbers. Indeed, accuracy can be obtained with floating point numbers, provided that we are less *naive*, use the appropriate theory and algorithms, and perform the computations with tested softwares.

6 Acknowledgments

I would like to thank Bruno Pinçon who made many highly valuable numerical comments on the section devoted to numerical derivatives. I would like to thank Claude Gomez for his support during the writing of this document. I would like to thank Bernard Hugueney and Allan Cornet who helped my in the analysis of the complex division portability issues.

I thank Robert L. Smith for his comments on the complex division section of this document.

7 Appendix

In this section, we analyze the examples given in the introduction of this article. In the first section, we analyze how the real number 0.1 is represented by a double precision floating point number, which leads to a rounding error. In the second section, we analyze how the computation of $\sin(\pi)$ is performed. In the final section, we make an experiment which shows that $\sin(2^{10i}\pi)$ can be arbitrary far from zero when we compute it with double precision floating point numbers.

7.1 Why 0.1 is rounded

In this section, we present a brief explanation for the following Scilab session. It shows that the mathematical equality $0.1 = 1 - 0.9$ is not exact with binary floating point integers.

```
-->format(25)
-->x1=0.1
 x1  =
    0.1000000000000000055511
-->x2 = 1.0-0.9
 x2  =
    0.0999999999999999777955
```

```
-->x1==x2
 ans  =
  F
```

We see that the real decimal number 0.1 is displayed as 0.100000000000000005. In fact, only the 17 first digits after the decimal point are significant : the last digits are a consequence of the approximate conversion from the internal binary double number to the displayed decimal number.

In order to understand what happens, we must decompose the floating point number into its binary components. The IEEE double precision floating point numbers used by Scilab are associated with a radix (or basis) $\beta = 2$, a precision $p = 53$, a minimum exponent $e_{min} = -1023$ and a maximum exponent $e_{max} = 1024$. Any floating point number x is represented as

$$fl(x) = M \cdot \beta^{e-p+1}, \tag{268}$$

where

- e is an integer called the exponent,
- M is an integer called the integral significant.

The exponent satisfies $e_{min} \leq e \leq e_{max}$ while the integral significant satisfies $|M| \leq \beta^p - 1$.

Let us compute the exponent and the integral significant of the number $x = 0.1$. The exponent is easily computed by the formula

$$e = \lfloor \log_2(|x|) \rfloor, \tag{269}$$

where the $\log_2$ function is the base-2 logarithm function. In the case where an underflow or an overflow occurs, the value of e is restricted into the minimum and maximum exponents range. The following session shows that the binary exponent associated with the floating point number 0.1 is -4.

```
-->format(25)
-->x = 0.1
 x  =
    0.10000000000000000055511
-->e = floor(log2(x))
 e  =
  - 4.
```

We can now compute the integral significant associated with this number, as in the following session.

```
-->M = x/2^(e-p+1)
 M  =
    7205759403792794.
```

Therefore, we deduce that the integral significant is equal to the decimal integer $M = 7205759403792794$. This number can be represented in binary form as the 53 binary digit number

$$M = 11001100110011001100110011001100110011001100110011010. \tag{270}$$

We see that a pattern, made of pairs of 11 and 00 appears. Indeed, the real value 0.1 is approximated by the following infinite binary decomposition:

$$0.1 = \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \dots \right) \cdot 2^{-4}. \tag{271}$$

We see that the decimal representation of $x = 0.1$ is made of a finite number of digits while the binary floating point representation is made of an infinite sequence of digits. But the double precision floating point format must represent this number with 53 bits only.

Notice that, the first digit is not stored in the binary double format, since it is assumed that the number is *normalized* (that is, the first digit is assumed to be one). Hence, the leading binary digit is *implicit*. This is why there is only 52 bits in the mantissa, while we use 53 bits for the precision p. For the sake of simplicity, we do not consider denormalized numbers in this discussion.

In order to analyze how the rounding works, we look more carefully to the integer M, as in the following experiments, where we change only the last decimal digit of M.

```
-->7205759403792793 * 2^(-4-53+1)
 ans  =
    0.099999999999999916733
-->7205759403792794 * 2^(-4-53+1)
 ans  =
    0.10000000000000000055511
```

We see that the exact number 0.1 is between two consecutive floating point numbers:

$$7205759403792793 \cdot 2^{-4-53+1} < 0.1 < 7205759403792794 \cdot 2^{-4-53+1}. \tag{272}$$

There are four rounding modes in the IEEE floating point standard. The default rounding mode is *round to nearest*, which rounds to the nearest floating point number. In case of a tie, the rounding is performed to the only one of these two consecutive floating point numbers whose integral significant is even. In our case, the distance from the exact x to the two floating point numbers is

$$|0.1 - 7205759403792793 \cdot 2^{-4-53+1}| = 8.33 \cdots 10^{-18}, \tag{273}$$

$$|0.1 - 7205759403792794 \cdot 2^{-4-53+1}| = 5.55 \cdots 10^{-18}. \tag{274}$$

(The previous computation is performed with a symbolic computation system, not with Scilab). Therefore, the nearest is $7205759403792794 \cdot 2^{-4-53+1}$, which leads to $fl(0.1) = 0.10000000000000005$.

On the other side, $x = 0.9$ is also not representable as an exact binary floating point number (but 1.0 is exactly represented). The floating point binary representation of $x = 0.9$ is associated with the exponent $e = -1$ and an integral significant between 8106479329266892 and 8106479329266893. The integral significant which is nearest to $x = 0.9$ is 8106479329266893, which is associated with the approximated decimal number $fl(0.9) \approx 0.90000000000000002$.

Then, when we perform the subtraction "1.0-0.9", the decimal representation of the result is $fl(1.0) - fl(0.9) \approx 0.09999999999999997$, which is different from $fl(0.1) \approx 0.10000000000000005$.

We have shown that the mathematical equality $0.1 = 1 - 0.9$ is not exact with binary floating point integers. There are many other examples where this happens. In the next section, we consider the sine function with a particular input.

7.2 Why $sin(\pi)$ is rounded

In this section, we present a brief explanation of the following Scilab 5.1 session, where the function sinus is applied to the number π.

```
-->format(10)
 ans  =
    0.
-->sin(%pi)
 ans  =
    1.225D-16
```

This article is too short to make a complete presentation of the computation of elementary functions. The interested reader may consider the direct analysis of the Fdlibm library as very instructive [47]. Muller presents in "Elementary Functions" [36] a complete discussion on this subject.

In Scilab, the sin function is connected to a fortran source code (located in the *sci_f_sin.f* file), where we find the following algorithm:

```
do i = 0 , mn - 1
  y(i) = sin(x(i))
enddo
```

The `mn` variable contains the number of elements in the matrix, which is stored as the raw array `x`. This implies that no additionnal algorithm is performed directly by Scilab and the sin function is computed by the mathematical library provided by the compiler, i.e. by gcc under Linux and by Intel's Visual Fortran under Windows.

Let us now analyze the algorithm which is performed by the mathematical library providing the sin function. In general, the main structure of these algorithms is the following:

- scale the input x so that in lies in a restricted interval,
- use a polynomial approximation of the local behavior of sin in the neighborhood of 0.

In the Fdlibm library for example, the scaling interval is $[-\pi/4, \pi/4]$. The polynomial approximation of the sin function has the general form

$$sin(x) \approx x + a_3 x^3 + \ldots + a_{2n+1} x^{2n+1} \quad (275)$$

$$\approx x + x^3 p(x^2) \quad (276)$$

In the Fdlibm library, 6 terms are used.

For the `atan` function, which is used to compute an approximated value of π, the process is the same. This leads to a rounding error in the representation of π which is computed by Scilab as `4*atan(1.0)`. All these operations are guaranteed with some precision, when applied to a number in the scaled interval. For inputs outside the scaling interval, the accuracy depends on the algorithm used for the scaling.

All in all, the sources of errors in the floating point computation of $\sin(\pi)$ are the following

- the error of representation of π,
- the error in the scaling,
- the error in the polynomial representation of the function sin.

The error of representation of π by a binary floating point number is somewhat hidden by the variable name `%pi`. In fact, we should really say that `%pi` is the best possible binary double precision floating point number representation of the mathematical π number. Indeed, the exact representation of π would require an infinite number of bits. This is not possible, which leads to rounding.

In fact the exact number π is between two consecutive floating point numbers:

$$7074237752028440 \cdot 2^{1-53+1} < \pi < 7074237752028441 \cdot 2^{1-53+1}. \quad (277)$$

In our case, the distance from the exact π to its two nearest floating point numberss is

$$|0.1 - 7074237752028440 \cdot 2^{1-53+1}| = 1.22 \cdots 10^{-16}, \quad (278)$$
$$|0.1 - 7074237752028441 \cdot 2^{1-53+1}| = 3.21 \cdots 10^{-16}. \quad (279)$$

(The previous computation is performed with a symbolic computation system, not with Scilab). Therefore, the nearest is $7074237752028440 \cdot 2^{1-53+1}$. With a symbolic computation system, we find:

$$\sin(7074237752028440 \cdot 2^{1-53+1}) = 1.22464679914735317e - 16. \quad (280)$$

We see that Scilab has produced a result which has the maximum possible number of significant digits.

We see that the rounding of `%pi` has a tremendous effect on the computed value of `sin(%pi)`, which is clearly explained by the condition number of this particular computation.

The condition number of the a smooth single variable function $f(x)$ is $c(x) = |xf'(x)/f(x)|$. For the sine function, this is $c(x) = |x\cos(x)/\sin(x)$. This function is large when x is large or when x is an integer multiple of π. In the following session, we compute the condition number of the sine function at the point `x=%pi`.

```
-->abs(%pi*cos(%pi)/sin(%pi))
 ans  =
    2.565D+16
```

With such a large condition number, any change in the last significant bit of `x=%pi` changes the first significant bit of `sin(x)`. This explains why computing `sin(%pi)` is numerically challenging.

7.3 One more step

In fact, it is possible to reduce the number of significant digits of the sine function to as low as 0 significant digits. We mathematical have $\sin(2^n\pi) = 0$, but this can be very inaccurate with floating point numbers. In the following Scilab session, we compute $\sin(2^{10i}\pi)$ for $i = 1$ to 5.

```
-->sin(2.^(10*(1:5)).*%pi).'
 ans  =
  - 0.00000000000001254038322
  - 0.00000000012840928320660
  - 0.00000013149110600352250
  - 0.00013466489214075421410
  - 0.13744197770626359611510
```

For $\sin(2^{50}\pi)$, the result is very far from being zero. This computation may sound *extreme*, but it must be noticed that it is inside the IEEE double precision range of values, since $2^{50} \approx 3.10^{15} \ll 10^{308}$. If accurate computations of the sin function are required for large values of x (which is rare in practice), the solution may be to use multiple precision floating point numbers, such as in the MPFR library [35, 14], based on the Gnu Multiple Precision library [17].

References

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

[2] Maxima. http://maxima.sourceforge.net/.

[3] Octave. http://www.gnu.org/software/octave.

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

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

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

[7] The Scilab Consortium. Scilab. http://www.scilab.org.

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

[9] 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.

[10] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. volume 1: Basic architecture. http://www.intel.com/products/processor/manuals, 2009.

[11] J. Dumontet and J. Vignes. Détermination du pas optimal dans le calcul des dérivées sur ordinateur. *R.A.I.R.O Analyse numérique*, 11(1):13–25, 1977.

[12] 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.

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

[14] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[15] Paul Friedland. Algorithm 312: Absolute value and square root of a complex number. *Commun. ACM*, 10(10):665, 1967.

[16] P. E. Gill, W. Murray, and M. H. Wright. *Practical optimization*. Academic Press, London, 1981.

[17] GMP. Gnu multiple precision arithmetic library. http://gmplib.org.

[18] 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.

[19] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

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

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

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

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

[24] M. A. Jenkins and J. F. Traub. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM Journal on Numerical Analysis*, 7(4):545–566, 1970.

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

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

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

[28] C. T. Kelley. *Solving nonlinear equations with Newton's method*. SIAM, 2003.

[29] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third Edition, Addison Wesley, Reading, MA, 1998.

[30] 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.

[31] 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.

[32] Maple. Maplesoft. http://www.maplesoft.com.

[33] The MathWorks. Matlab. http://www.mathworks.fr.

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

[35] MPFR. The mpfr library. http://www.mpfr.org.

[36] Jean-Michel Muller. *Elementary functions: algorithms and implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 1997.

[37] 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.

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

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

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

[41] Wolfram Research. Mathematica. http://www.wolfram.com/products/mathematica.

[42] Wolfram Research. Wolfram alpha. http://www.wolframalpha.com.

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

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

[45] R. S. Stepleman and N. D. Winarsky. Adaptive numerical differentiation. *Mathematics of Computation*, 33(148):1257–1264, 1979.

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

[47] Inc. Sun Microsystems. A freely distributable c math library. 1993. http://www.netlib.org/fdlibm.

[48] P. Wynn. An arsenal of ALGOL procedures for complex arithmetic. *BIT Numerical Mathematics*, 2(4):232–255, December 1962.

Index