Informatics in Education, 2003, Vol. 2, No. 1, 21–38

Numerical Difficulties in Pre-University Informatics Education and Competitions

Gyula HORVÁTH
University of Szeged, Hungary
e-mail: horvath@inf.u-szeged.hu

Tom VERHOEFF
Eindhoven Univ. of Techn., The Netherlands
e-mail: t.verhoeff@tue.nl

Received: March 2003

Abstract. It is easy to underestimate the difficulties of using floating-point numbers in programming. This is especially the case in pre-university informatics education and competitions, where one is often led to believe that floating-point arithmetic is a good approximation of the real number system. However, most of the mathematical laws valid for real numbers break down when applied to floating-point numbers. We explain this break-down and illustrate it with four simple examples.

In informatics education and competitions, the students need to be trained, programming assignments need to be formulated, submitted programs need to be evaluated, and variations among computing platforms need to be handled. We show that the use of floating-point numbers gives rise to various kinds of non-trivial difficulties in all these areas. Coping with such difficulties would require that teachers, students, and organizers gain experience in numerical mathematics.

We strongly recommend to avoid the use of floating-point numbers in pre-university education and competitions whenever possible. If you do want to use floating-point numbers, then study the literature of numerical mathematics and be prepared to do a convincing error analysis.

Key words: informatics education, informatics competitions, floating-point arithmetic, numerical mathematics.

1. Introduction

Non-integer numbers are introduced in mathematics education soon after pupils are familiar with the integers. In primary education, pupils learn to deal with fractions and percentages, and with fixed-point currency values. In secondary education, they also encounter the real number system as closure of the rational numbers. The square root of two is a real number, but not a fraction (of integer numbers). These real numbers play an important role in all sciences, especially physics, where continuous mathematical models are used.

In many science classes, pocket calculators are used to help solve numerical problems. For that purpose, these calculators must support the so-called *scientific notation*, where a number is expressed as the product of a number between 1 and 10 (the decimal portion), and a power of 10 (the exponential portion). The motivation usually given for this notation is that it simplifies the handling of very small (near zero) and very large numbers. Often less well explained is the notion of approximation and significance.

Every serious programming language provides floating-point numbers to tackle numerical problems involving non-integer calculations. These floating-point numbers are the computer's scientific notation. For instance, Pascal has the type *real*, and C, C++, and Java have the type *float*. In fact, there are often several floating-point types to choose from.

For some people, the obvious conclusion from these observations seems to be that programming education at the pre-university level must include the use of floating-point numbers. Often, they presume that floating-point numbers can be treated like integers when designing programming exercises and competition problems.

In this paper, we point out various kinds of difficulties that arise when using floating-point numbers. Some of these difficulties are fairly straightforward, but others are quite subtle. It is important to understand these difficulties if one wishes to maintain a high quality standard in science education. As an omen, we quote D. E. Knuth (Knuth, 1997):

> "Floating point computation is by nature inexact, and programmers can easily misuse it so that the computed answers consist almost entirely of "noise." One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be. There is a "credibility-gap": We don't know how much of the computer's answers to believe. Novice computer users solve this problem by implicitly trusting in the computer as an infallible authority; they tend to believe that all digits of a printed answer are significant. Disillusioned computer users have just the opposite approach; they are constantly afraid that their answers are almost meaningless."

1.1. *Overview*

In the remainder of this article, the terms 'floating-point' and 'real' can often be replaced by the more general term 'non-integer'. Special cases of non-integer numbers include fixed-point and rational numbers. Many of the concerns about floating-point numbers apply to those as well.

We start with four programming examples involving floating-point numbers. The next section presents some of the basic concepts and terminology of numerical mathematics. Then we analyze the examples and point out various difficulties, introducing further concepts where needed. In Section 5, we investigate the difficulties encountered when using floating-point numbers in education and competitions, especially in the International Olympiad in Informatics. Because the literature on numerical mathematics is so vast, we have included some guidance in Section 6. We conclude the paper with some recommendations.

2. Four Examples

The following examples show simple computing situations in which floating-point numbers may cause trouble. The actual difficulties will be explained in Section 4.

Count down. The first example involves the following program, here shown in Pascal (left) and C (right):

```
const D = 0.1;
var x: Real;
begin
  x := 1.0;
  while x > 0.0 do
    x := x - D;
  writeln ( x:1:2 )
end.
```

```
#include <stdio.h>
#define D 0.1
int main ( void )
{ double x = 1.0;
  while ( x > 0.0 )
    x = x - D;
  printf ( "%1.2f\n", x );
}
```

Can you tell what value this program prints, without running it? The result may depend on your computing platform. You can experiment with the type of variable x (single versus double), and the value of constant D (0.1 versus 0.01).

Euclidean paths. For the second example, consider these four points in 3D space:

A	B	C	D
$(2, 5, 31)$	$(1, 2, 9)$	$(0, 7, 27)$	$(1, 8, 10)$

(1)

All these points have innocent coordinates: small integers. Consider the two V-shaped paths via the origin O: AOB and COD. Are the lengths of these two paths equal? If not, which is bigger? Now also tackle the case with

A	B	C	D
$(4, 12, 28)$	$(2, 6, 14)$	$(1, 1, 23)$	$(1, 13, 19)$

(2)

Parallel resistors. The third example concerns the computation of the effective resistance of two parallel resistors (see Fig. 1). Write a program to compute the effective resistance, given the non-negative values R_1 and R_2 as input.

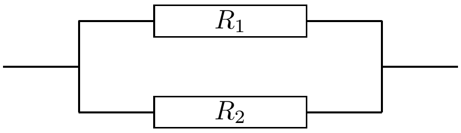

Fig. 1. Two parallel resistors: what is the effective resistance?

Quadratic equation. As our fourth example we consider the equation

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

where parameters a, b, and c are given real constants and x is a real variable whose value(s) satisfying (3) must be determined. Write down your choice of conditions to impose on the parameters to make this into a practical programming assignment. You may decide whether to include the case analysis caused by zero parameters. Then solve your own assignment. Finally, explain how you would evaluate programs that are handed in to solve your assignment.

3. Numerical Mathematics

We will not present a complete introduction into the field of numerical mathematics. That would take up too much space, and others have already done so (see Section 6). However, it is important to know some of the concepts and terminology.

We denote the set of **real numbers** by $\mathbb{R}$, and by $\mathbb{F}$ its subset of **floating-point numbers** representable on our hypothetical computing platform. Note that $\mathbb{F}$ is a finite subset of the rational numbers and that it depends on the specifics of the computer hardware, operating system, and programming language (compiler). On PCs, $\mathbb{F}$ usually includes numbers x of the form

$$x = f \times 2^e, \tag{4}$$

where the **fraction** f and **exponent** e are numbers from a limited set: $f \times 2^t$ is an integer with $f = 0$ or $1 \leqslant |f| < 2$, and e is an integer with $e_{\min} \leqslant e \leqslant e_{\max}$, for appropriate integers t, $e_{\min}$, and $e_{\max}$. Note that terminology in the literature varies: some authors prefer to work with the 'mantissa' $f \times 2^t$ or the fraction $f/2$ instead of f, applying an appropriate shift to the exponent. The value $p = t + 1$ is the number of bits in the binary representation of f; it is called the **precision** of $\mathbb{F}$.

The number 1 is represented by $(f, e) = (1, 0)$. The smallest floating-point number larger than 1 is $1 + \epsilon$ with $\epsilon = 2^{-t}$ called the **machine epsilon** of $\mathbb{F}$. The smallest positive floating-point number is $N_{\min} = 2^{e_{\min}}$ and the largest one is $N_{\max} = (2 - \epsilon)2^{e_{\max}}$. Typical values of t, $e_{\min}$, $e_{\max}$, and ϵ are shown in Table 1. The interval $N_{\min}$ to $N_{\max}$ is called the **range** of $\mathbb{F}$. Note that $N_{\min} \ll \epsilon$.

Most operations on $\mathbb{R}$ are not closed in $\mathbb{F}$. When such operations are simulated on a computer, the result is forced into $\mathbb{F}$, yielding an **approximation** of the exact result. This may introduce a (small) **rounding error** into floating-point calculations. Subsequent operations on inexact results can magnify, or reduce, the error in non-intuitive ways. The aim of **error analysis** is to understand the propagation of errors in numerical algorithms, in particular to prove bounds on the error in the final result. Doing an error analysis is a constant struggle between being too optimistic and being too pessimistic.

Students generally assume that the floating-point numbers $\mathbb{F}$ form an adequate model of the real numbers $\mathbb{R}$. Especially, they assume that every identity valid for real numbers, like the associative law $(x + y) + z = x + (y + z)$, also holds for floating-point numbers. To express this assumption formally, let *fl*: $\mathbb{R} \to \mathbb{F}$ be an approximation function from the real numbers to the floating-point numbers. That is, *fl*(x) is the floating-point number nearest to real number x. For any computation $\mathcal{A}$: $\mathbb{R}^n \to \mathbb{R}$ with n real inputs and

Table 1

Parameters for normalized binary IEEE floating-point numbers

Type	Size	t	$e_{\min}$	$e_{\max}$	ϵ	Range
Single	32 bits	23	-126	127	$2^{-23} \approx 1.2 \times 10^{-7}$	$\approx 10^{\pm 38}$
Double	64 bits	52	-1022	1023	$2^{-52} \approx 2.2 \times 10^{-16}$	$\approx 10^{\pm 308}$

one real output we denote the corresponding machine computation by $\widehat{\mathcal{A}}\colon \mathbb{F}^n \to \mathbb{F}$. The assumption is that the following diagram commutes.

$$\begin{tikzcd}
\mathbb{R}^n \arrow[r, "fl^n"] \arrow[d, "\mathcal{A}"] & \mathbb{F}^n \arrow[d, "\widehat{\mathcal{A}}"] \\
\mathbb{R} \arrow[r, "fl"] & \mathbb{F}
\end{tikzcd} \tag{5}$$

This is equivalent to the requirement that the basic arithmetic operations $\diamond \in \{+, -, \times, /\}$ satisfy the equation

$$fl(x \diamond y) = fl(x) \mathbin{\widehat{\diamond}} fl(y), \tag{6}$$

for all real numbers x and y, where $\widehat{\diamond}$ denotes the machine implementation of the operation $\diamond$. In other words, *fl* is assumed to be a homomorphism from the algebra over the real numbers $\mathbb{R}$ to the algebra over the floating-point numbers $\mathbb{F}$. Unfortunately, this is not true. As a counterexample, imagine that our hypothetical machine works with two decimal digits only. We then have

$$fl(1.06 + 3.06) = fl(4.12) = 4.1 \neq 4.0 = 1.0 \mathbin{\widehat{+}} 3.0 = fl(1.06) \mathbin{\widehat{+}} fl(3.06).$$

The IEEE standard for floating-point arithmetic does require that

$$x \mathbin{\widehat{\diamond}} y = fl(x \diamond y), \tag{7}$$

for all floating-point numbers x and y. Since $fl(x) = x$ for $x \in \mathbb{F}$, this means that (6) at least holds in $\mathbb{F}$.

Even floating-point computations that yield a *boolean* value rather than a real value are to be suspected. This can be understood by considering the evaluation of the following comparison:

$$5.3 \times 0.2 + 5.1 \times 0.6 < 1.1 \times 1.9 + 5.1 \times 0.4. \tag{8}$$

The exact evaluation of the multiplications yields the (obviously true) inequality:

$$1.06 + 3.06 < 2.09 + 2.04. \tag{9}$$

On our hypothetical machine working with two-digit decimal numbers only, the four multiplication results are rounded before the additions are done. By approximation we would then obtain the (obviously false) inequality:

$$1.1 + 3.1 < 2.1 + 2.0. \tag{10}$$

Indeed, the left-hand side is now *larger* than the right-hand side. Note that the additions are done exactly on our hypothetical machine.

4. Analyzing the Four Examples

Let us look at the four examples in turn.

Count down. The value $D = 0.1$ cannot be represented exactly as a binary floating-point number, because $1/10$ has an infinite repeating binary representation:

$$0.000110011001100110011001100110011001100\ldots = \sum_{k=1}^{\infty} 3/2^{4k+1}. \tag{11}$$

Therefore, in the program we have $D = \mathit{fl}(0.1) \neq 0.1$. Subtracting D from 1.0 ten times in IEEE double precision yields approximately 10^{-16}, which still exceeds 0. The next subtraction yields -0.10 when printed with two decimals. Paradoxically, the program prints 0.00 when run with the less precise single format.

Such a loop should have been controlled by an integer, e.g., $i = 10 \cdot x$. In that case, x could even be eliminated by using $i/10$. A common beginner's mistake is to take $x \neq 0.0$ as looping condition, in which case the loop might never terminate.

Similar mistakes are the abuse of floating-point numbers for currency calculations, or for comparing fractions by writing $a/b = c/d$, or for testing divisibility by $\mathit{round}(a/b) = a/b$. Fractions are better compared by $a * d = b * c$ (however, overflow is a concern), and divisibility is better (and faster) tested by $a \ \mathbf{mod}\ b = 0$.

Euclidean paths. By applying Pythagoras' Theorem, we find for the lengths of the first pair of V-paths:

$$\begin{aligned} AOB &= \sqrt{990} + \sqrt{86} &\approx\ & 40.73788394060\ldots \\ COD &= \sqrt{778} + \sqrt{165} &\approx\ & 40.73788394062\ldots \end{aligned}$$

The two lengths coincide on the 12 most significant decimal digits, with a difference on the order of 10^{-11}. They are indistinguishable with single-precision computer calculation and on many pocket calculators. For the second pair we find

$$\begin{aligned} AOB &= \sqrt{944} + \sqrt{236} &\approx\ & 46.0868744487211645\ldots, \\ COD &= \sqrt{531} + \sqrt{531} &\approx\ & 46.0868744487211652\ldots, \end{aligned}$$

where the difference is less than 10^{-14}. Are the lengths really different?

For the first pair the answer is 'Yes', for the second 'No'. The verification can be carried out in the integer domain, with some care. For the second pair observe

$$\begin{aligned} \sqrt{944} + \sqrt{236} &= \sqrt{16 \cdot 59} + \sqrt{4 \cdot 59} &=\ & 6\sqrt{59}, \\ \sqrt{531} + \sqrt{531} &= \sqrt{9 \cdot 59} + \sqrt{9 \cdot 59} &=\ & 6\sqrt{59}. \end{aligned}$$

For the first pair, the assumption of equality leads to a contradiction by three squarings and subsequent simplifications:

$$\sqrt{990} + \sqrt{86} = \sqrt{778} + \sqrt{165},$$

$$\begin{aligned}
990 + 2\sqrt{990 \cdot 86} + 86 &= 778 + 2\sqrt{778 \cdot 165} + 165, \\
133 &= 2 \cdot \left(\sqrt{778 \cdot 165} - \sqrt{990 \cdot 86}\right), \\
133^2 &= 4 \cdot \left(778 \cdot 165 - 2\sqrt{778 \cdot 165 \cdot 990 \cdot 86} + 990 \cdot 86\right), \\
8\sqrt{778 \cdot 165 \cdot 990 \cdot 86} &= 4 \cdot (778 \cdot 165 + 990 \cdot 86) - 133^2, \\
64 \cdot 778 \cdot 165 \cdot 990 \cdot 86 &= 836351^2, \\
699482995200 &= 699482995201.
\end{aligned}$$

Note that 32-bit integer arithmetic does not suffice.

The original problem involves small integer coordinates. The results of floating-point calculations on these numbers must be interpreted carefully. Replacing them by integer calculations presents other complications. For larger coordinates or more than two segments in a path, even double-precision calculations do not suffice, and a reduction to integer calculations is a nightmare. This just is *not* an easy problem. It is difficult to determine safe bounds on the size of the coordinates and the number of segments such that floating-point computations can be used naïvely to pick the longer path.

The competition task BUS TERMINALS at IOI 2002 (Day 2) involves a set of points in the plane with positive integer coordinates at most 5000 (IOI2002, 2002). The goal is to select two points A and B (called 'hubs') from the set and to assign each of the remaining points to either A or B while minimizing a certain cost function (see Fig. 2). The cost $c(P, Q)$ associated with two points P and Q is defined by

$$c(P, Q) = d(P, H(P)) + d(H(P), H(Q)) + d(H(Q), Q), \tag{12}$$

where d measures the distance between points in the plane and $H(X) \in \{ A, B \}$ is the hub assigned to point X (for convenience, $H(A) = A$ and $H(B) = B$). The cost to be minimized is the maximum value of $c(P, Q)$ over all pairs of distinct points in the given set.

This task uses the Manhattan distance for d to avoid numerical complications. An earlier formulation was based on the Euclidean distance, which would then require the comparison of sums of two or three square roots of integers having up to eight decimal digits. It is far from obvious that double-precision floating-point calculations would suffice, and it might simply be incorrect. Even if the organizers would have proved that double-precision suffices, then it is still questionable whether the contestants could have discovered this. For reasons of speed, a contestant might be tempted to use single precision, and get away with it because the test cases used for evaluation happen to miss

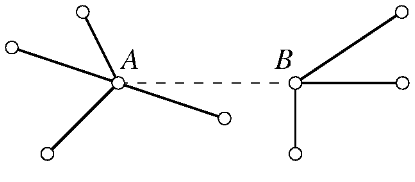

Fig. 2. Example assignment of points for task BUS TERMINALS.

the bad cases where single precision fails. Stating in the task description that double-precision floating-point suffices to calculate and compare costs (if this would indeed be true), might still get contestants into trouble if they apply some simple transformations to the cost function (see Quadratic equation below). Because the actual distance function does not matter much for the algorithmic aspects of the task, it was decided to use an integer-based distance function, thereby avoiding floating-point numbers completely.

Parallel resistors. Recall that the replacement resistance R for two parallel resistors R_1 and R_2 is given by

$$R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}. \tag{13}$$

However, this formula cannot be used directly when $R_1 = 0$ or $R_2 = 0$, even though R is well defined in those cases. The program needs to do a case analysis. Rewriting the formula as

$$R = \frac{R_1 \cdot R_2}{R_1 + R_2} \tag{14}$$

helps somewhat, but still is problematic when $R_1 = 0 = R_2$. Furthermore, such rewriting quickly becomes a mess when more than two resistors are involved. On the other hand, if your computing platform supports the IEEE floating-point standard, then the first formula can indeed be used, but not the second. The IEEE floating-point standard includes well-behaved **infinities** with $1/0 = \infty$, $\infty + x = \infty$, $1/\infty = 0$ (actually, zeroes and infinities are signed). However, $0/0$ is not defined, yielding a **NaN** (not-a-number) in IEEE terminology. To use this standard properly requires some studying. Unfortunately, many programming languages do not (fully) support the IEEE standard.

Quadratic equation. In high school, the a, b, c-formula is taught for solving quadratic equations:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{15}$$

In programming classes it is often used to practice the translation of a mathematical formula into a program expression. Numerically speaking, however, the a, b, c-formula is not appropriate in many cases. If you apply it to the equation

$$10^{-8} \times x^2 + x - 1 = 0, \tag{16}$$

and evaluate it in IEEE single precision, you obtain the two roots

$$x_{1,2} = 0.000000000, \quad -1.000000000 \times 10^8. \tag{17}$$

The true positive root is actually 1.0 with an accuracy better than 10^{-7}. Thus, the positive root computed from the a, b, c-formula is off by 100%. The reason for this large error is that in the a, b, c-formula for our positive root, the values $-b$ and $+\sqrt{b^2 - 4ac}$ have opposite signs and are almost equal in size, because $|4ac| \ll b^2$. When adding them, the (roundoff) error present in the computed value for $b^2 - 4ac$ is suddenly magnified enormously in relative size. This phenomena is known as **cancellation**. Here, it can be avoided by multiplying the numerator and denominator in the a, b, c-formula by $-b \mp \sqrt{b^2 - 4ac}$ and simplifying the result, yielding

$$x_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \tag{18}$$

This less-known formula is numerically more suitable in some cases, though there are even better approaches (e.g., viewing equation (16) as linear).

> **Sidenotes**
>
> Equation (16) may seem quite special, with $a \approx 0$. But this is not so. It could easily arise in practice. Besides, the point is to show that a program for solving quadratic equations should not apply the a, b, c-formula carelessly. The formula is correct, but it has numerical shortcomings.
>
> Furthermore, cancellation occurs whenever $|4ac| \ll b^2$, also when a is not close to zero. It then always gives rise to excessive loss of accuracy. Our example was chosen because of the extreme loss of accuracy: seven orders of magnitude. This is also the case when using double precision, except that you need to look more carefully!
>
> When experimenting with this equation, one must ensure that single precision is used for all calculations, especially for the subtraction $b^2 - 4ac$. When done in higher precision, another approximation for the positive root is found, but is still seven orders less accurate than possible at that precision.

The **accuracy** of an approximation can be expressed in terms of the error with respect to the exact value. Suppose the exact value $x \in \mathbb{R}$ is approximated by $\widehat{x} \in \mathbb{F}$ through some calculation. The **absolute error** (in $\widehat{x}$ for x) is defined as

$$|x - \widehat{x}|. \tag{19}$$

Another measure is the **relative error**, defined as

$$\frac{|x - \widehat{x}|}{|x|}. \tag{20}$$

Scientific and engineering applications often involve scaling, e.g., when converting values to other units. The relative error is preferred because it is **invariant under scaling**. It is, however, not defined when $x = 0$. That makes it inappropriate when the outcome can be exactly zero.

An accuracy requirement is typically expressed as a bound on the (absolute or relative) error. Every real number can be approximated by an IEEE single-precision floating-point number with a relative error less than 10^{-7}, provided it lies within range. Therefore, an error of 100% in the positive root seems extremely bad. The alternative formula (18) shows that one can indeed do much better. Even though (15) and (18) are mathematically identical (in $\mathbb{R}$), they behave very differently in a numeric sense (in $\mathbb{F}$). This illustrates how dramatic the mismatch between the algebra of real numbers and that of floating-point numbers can be.

A numerical algorithm is called **stable**, when it produces answers whose accuracy is on the order of what can 'reasonably' be expected for the problem at hand. For the positive root of (16), the a, b, c-formula (15) is unstable, whereas the alternative formula (18) is stable. One of the challenges in numerical mathematics is to determine what can 'reasonably' be expected and to find appropriate stable algorithms.

Unfortunately, the cancellation mentioned above is not the whole story. Cancellation is also possible in the subtraction $b^2 - 4ac$ when $b^2 \approx 4ac$. In this case it is harder to circumvent, because it is inherent in the problem itself and not a consequence of a badly chosen algorithm. The parabola is then almost tangent to the line $y = 0$. Determining the roots when they are nearly equal is said to be an **ill-conditioned problem**.

Furthermore, the squaring b^2, the multiplication $4ac$, and the final division by $2a$ can produce (intermediate) results that fall outside the representable range. This is referred to as **underflow** or **overflow**. For b^2 and $4ac$ this can happen even if the final results are representable within the range of floating-point numbers.

When setting the solution of the quadratic equation as a programming assignment, one needs to worry about

1) the restrictions on the input coefficients a, b, c,
2) what to do in case of roots that are not representable within the range of floating-point numbers,
3) what to do in case of complex roots,
4) the desired accuracy of the output roots,
5) how to evaluate a quadratic-solving program.

In a high-school setting, one might try to restrict the coefficients such that

- the roots are always real and representable;
- a 'reasonable' approach does not suffer from overflow or underflow in intermediate results;
- the problem is not ill-conditioned.

It takes a careful analysis to determine such safe restrictions, and they are either awkward to formulate, or yield a crippled problem.

The quadratic-solving program becomes much more complicated if it explicitly needs to deal with the possibility of intermediate overflows and underflows, and unrepresentable roots. It is extremely difficult to determine the reliability of such programs merely by testing with specific cases. The burden of proof should be on the designer, who must provide a convincing argument that the program satisfies the specification.

5. Using Floating-Point Numbers at the Pre-University Level

In this section, we investigate the difficulties of using floating-point numbers in pre-university informatics education and competitions. We will consider the following common aspects:

1. Statement of programming tasks.
2. Evaluation of submitted programs.
3. Preparation of students.
4. Choice of platform (CPU, OS, compiler).

We will focus on the International Olympiad in Informatics, rather than a general educational setting, because it provides a well-defined context.

The International Olympiad in Informatics (IOI) is an annual competition in computing science (informatics) for talented high-school students from all over the world (IOI, 2003). Further information can also be found in (Horváth and Verhoeff, 2002). The IOI competition tasks are typically algorithmic in nature. The contestants have to design algorithms that satisfy precise specifications in terms of input, output, and performance constraints. However, it is not a paper-and-pen exam. These algorithms must also be implemented in one of the allowed programming languages[1].

IOI competition tasks have traditionally been formulated in terms of integers and (character) strings, avoiding the use of floating-point numbers. The task descriptions contain a precise specification of bounds on the inputs. These bounds are chosen in such a way that the resulting problem can be solved within the stated time and memory limits, and without worrying too much about input/output formats and integer overflow. Competitors can treat their computer as a good approximation of an ideal machine as far as integer arithmetic is concerned.

5.1. *Stating the Tasks*

Various kinds of floating-point tasks can be distinguished. First there is the distinction based on visibility:

implicit Tasks where neither inputs nor outputs involve floating-point numbers, but for which the organizer's preferred solution does involve them (e.g., because it is much faster or easier than a program based on integer arithmetic only).

explicit Tasks where inputs or outputs do involve floating-point numbers. It is relevant to distinguish the three subcases where

- both inputs and outputs include floating-point numbers;
- inputs include floating-point numbers, but outputs not;
- outputs include floating-point numbers, but inputs not.

[1] Currently, Pascal, C, and C++ are supported.

When using the Euclidean distance, the task BUS TERMINALS of IOI2002 (also see Euclidean paths above) classifies as implicit, provided that the bounds are chosen such that floating-point arithmetic suffices. The task of solving a quadratic equation classifies as explicit, if the coefficients and roots are not specifically restricted to integers.

A second distinction might be based on the degree of numerical insight needed to obtain a perfect score:

tame Tasks where no numerical difficulties actually arise, that is, competitors can treat their computer (almost) as if it were an ideal machine.

mild Tasks where some 'obvious' 'standard' numerical techniques play a role.

wild Tasks requiring a more innovative approach to overcome the numerical difficulties.

Task TRIANGLES at BOI 2002 (BOI2002, 2002) is a nice example of a (nearly) tame task with integer inputs and explicit fixed-point output. It involves unions of triangles, whose total area is always of the form $n/2$, where n is an integer. These areas can be represented exactly as binary floating-point numbers[2], provided the range and precision are large enough. However, in order to decide what type to use for accumulating the total area, one needs to know about the details of the floating-point types available in the programming language of choice. The IEEE single format does not suffice, because the shape could consist of 2000 disjoint triangles of area $499\,000.5$ each. The total area then is $998\,001\,000.0 \approx 10^9 \approx 2^{30}$, which has 27 significant bits, thereby exceeding the 24 bits precision provided by the single format.

Therefore, the task TRIANGLES is not truly tame. The best approach is *not* to view it as a task involving floating-point numbers, and to calculate twice the total area in (32-bit) integers. The output can be written in two steps using **div** for the integer part, and **mod** for deciding between a trailing `'.0'` and `'.5'`.

We conjecture that floating-point numbers can easily be eliminated from tame tasks without affecting the algorithmic challenge. In the task TRIANGLES above, the required output could have been twice the total area.

Even in a seemingly tame task, one cannot be completely oblivious of numerical issues, as is shown by task TRIANGLES where the addition of many exact numbers fails if the wrong floating-point type is used. Furthermore, simple transformations valid in $\mathbb{R}$ may fail badly in $\mathbb{F}$. It would be pedagogically wrong to blame students for applying such transformations. Thus, tame tasks are actually at least mild, the 'standard' technique being that you must know when *not* to use floating-point arithmetic and when *not* to transform certain expressions.

Mild tasks – if they exist – would not make good IOI tasks because they assume specific knowledge, namely about 'standard' numerical techniques. This is counter to the spirit of the olympiad.

The actual task description also poses various dilemmas:

[2] This would no longer be the case for a generalization to three dimensions, where a coefficient of $1/3$ appears.

- Does it explicitly state in which class (tame, mild, or wild) the task falls? Does it help in the selection of appropriate floating-point type (single versus double)?
- How does it express what is known about the inputs: format details, value restrictions (such as bounds), and (when applicable) accuracy?
- How does it express what is required of the outputs: format details and accuracy?

For example, when task BUS TERMINALS is posed with Euclidean distance, how should a competitor know whether to use integers, single precision, or double precision? Concerning accuracy, we feel that inputs should always be given as exact values, that is, without error. It is clearly problematic that input values are typically presented in decimal notation, because after reading them into a floating-point variable, there usually is already a conversion error. Think of reading in the step value 0.1 in the Count-down program discussed earlier. To specify the accuracy of output values, there is a dilemma of bounding the absolute or the relative error.

Another question to be addressed is: How to convince reviewers and leaders that all allowed inputs are 'reasonably' solvable. This requires a careful numerical error analysis and possibly a lot of experimentation. Even understanding such an analysis requires a thorough familiarity with numerical mathematics. There is especially a concern when the input and/or internal calculations involve floating-point arithmetic, but the output is discrete. As an example, we have seen task BUS TERMINALS of IOI2002.

5.2. *Evaluating the Programs*

Let us assume that a floating-point task can be formulated in a satisfactory way. How will submitted programs for that task be evaluated? Since 1994, programs at the IOI are automatically evaluated based on the program's behavior for a small set of test cases.

Especially for mild and wild tasks, there is the problem that it may be very hard to find good test cases that separate sloppy programs from scientifically well-designed programs. For example, a program using single-precision floating-point numbers for task TRIANGLES above, would successfully solve many cases, in spite of being fundamentally flawed. The organizers would have to construct specific cases to detect floating-point flaws in such a program. In numerical mathematics, one requires convincing documentation for such programs. It is almost impossible to determine the quality of a numerical program by doing black-box tests.

If the task requirements are such that the input values uniquely determine the output values, then the outputs of a submitted program can be checked by a **syntactic comparison** to the know correct outputs. Tasks involving floating-point outputs might not have this property. In that case, the output check must be based on a **semantic comparison**. One must now take into account the conversion errors. Assume that the submitted program computes a floating-point value in variable r and writes it to a file in decimal notation, which the checker subsequently reads from that file into a floating-point variable s. Because of the unavoidable inaccuracies in the binary-to-decimal and decimal-to-binary, it is very well possible that $r \neq s$. Try this simple experiment: write *sqrt*(2) to a file, read the written value back, and compare it to the original value.

When exact outputs cannot be determined (e.g., analytically), there is also a technical problem: how to check the outputs of submitted programs? Assume that the accuracy requirement in the task description specifies an absolute error at most $\delta > 0$. Let x be the exact value of the solution, let y be the approximation produced by the organizer's program to be used for evaluation, and let z be the value output by the submitted program. It is tempting to accept or reject z depending on whether or not it satisfies:

$$|y - z| \leqslant \delta. \tag{21}$$

For the sake of the argument suppose that $x < y$. Now consider two outputs $z_1 = x - \delta$ and $z_2 = y + \delta$ of two submitted programs. Then (21) will reject z_1 and accept z_2, even though $|x - z_1| \leqslant \delta$ and $|x - z_2| > \delta$. This is clearly not satisfactory. One might try to solve this by checking for

$$|y - z| \leqslant \delta + \delta', \tag{22}$$

where δ' is a bound on the absolute error in value y computed by the organizers. The disadvantage of this alternative is that it may now accept outputs like z_2 that do not satisfy the accuracy requirement of the task description. This is also not a desirable situation. The least one can do as an organizer is determine y such that $\delta' \ll \delta$. This means that the program to produce such y's must be designed for a much tighter error bound than specified in the task description.

Another questions is: How to convince reviewers and leaders that the test cases and their evaluation are reasonable.

5.3. *Preparing the Students*

Even in the current situation, where the IOI does not involve floating-point tasks, competitors must be educated to some extent. For instance, they must be strongly advised to avoid floating-point numbers.

When floating-point tasks would be used, it is necessary to educate competitors better. For instance, they should at least understand that most mathematical theorems about rational and real numbers fail when converted to floating-point arithmetic. It may be difficult, however, to decide what standard numerical techniques to teach them in preparation for mild or wild tasks. This prerequisite knowledge is not so easy to codify as it may seem.

Furthermore, if you teach them ‘too much’, they may worry too much when dealing with tame or mild tasks. Assume a tame task is posed without explicitly stating that it is tame. Then a competitor who knows about mild and wild tasks may spend some time on analyzing the task description to determine that it is tame. Tameness need not be obvious at all. On the other hand, a naïve competitor (not knowing about numerical difficulties), simply goes ahead. This is not fair. The naïve competitor is in fact bluffing, and this should not be encouraged.

Alternatively, the numerically-educated competitor might avoid the analysis and decide to assume the worst, i.e., that it is a mild or even wild task. Consequently, he applies numerical techniques to guarantee that the outputs satisfy the requirements (no matter what). This involves extra work (costs time) and may not be without penalty (slower program using more memory) or risk (more error-prone). As an example, consider the complications when avoiding floating-point numbers in task BUS TERMINALS using the Euclidean distance.

Either the tameness must be explicitly stated in the task description, or it must otherwise be obvious from the task description (if the task is indeed intended as tame). The reviewers and leaders must also be convinced that it is indeed obviously a tame task.

5.4. *Handling the Platform Limitations*

The IOI involves various platforms[3] [4]). The same algorithm implemented on different platforms may give rise to significantly different outputs for the same inputs. This is especially a concern when competitors develop their programs on a platform that differs from the platform used for evaluation. The evaluation of programs needs to take this into account.

The IEEE Standard for binary floating-point arithmetic is not equally supported in all languages used at the IOI. There are also the floating-point specifics of the processors and compilers used in the competition. Often, these offer additional features outside the IEEE Standard. We will not delve into details here. This may create quite different challenges depending on the platform. These differences must be well understood when deciding about the suitability of a competition task.

6. Notes on Literature

Much of the literature on numerical mathematics involves calculus and advanced linear algebra. It often discusses the numerical solution of problems that are outside the scope of pre-university education, such as the approximation of special functions, matrix inversion, and integration of partial differential equations. We give a brief overview of accessible literature on numerical mathematics.

For almost any computer science problem it is good to consult the works of Donald Knuth. Chapter 4 (Arithmetic) of (Knuth, 1997) is worth reading. In particular, §4.2.1 and §4.2.2 contain many details on floating-point arithmetic and its accuracy. Unfortunately, treatment of the IEEE standard has been postponed until the fourth edition.

The official definition of the IEEE standard for binary floating-point arithmetic is (IEEE, 1985). Good expositions of this standard are available in (Goldberg, 1991; Higham, 1996; Overton, 2001). William Kahan is the academic father of this standard. His paper (Kahan, 1981) motivates the need for the standard, and in (Kahan, 1996) he

[3] Currently, Linux and Windows are offered for development; Linux is used for evaluation.

[4] At IOI 2002, FreePascal 1.0.6 and GCC 2.95.2 were available

explains some of the deficiencies in the current support for the standard, especially the lack of full support in many compilers.

A light introduction to numerical computing is (Overton, 2001). Well known and very practical are books from the series *Numerical Recipes*, such as (Press *et al.*, 1989), though some people have warned for shortcomings. The C version is available on-line, and we recommend §1.3 about error, accuracy, and stability. A more theoretical treatment can be found in (Higham, 1996). The first two chapters contain many inspiring examples. It also has good notes for references through its extensive bibliography.

Our examples for the sums of square roots (Euclidean paths) are based on (Goddijn, 2002). Comparing sums of square roots of integers is a known-difficult problem with applications in computational geometry (O'Rourke, 1981). It is unknown how much precision is needed to decide which of two such sums is larger, expressed in terms of the number of square roots and the size of the integers. Ron Graham constructed an example of two such sums with nine terms each and integers of seven decimal digits that agree up to 40 decimal digits (Graham, 2003).

The quadratic equation is covered in depth in §9.6 of (Sterbenz, 1974). Also see §1.8 and §1.21 in (Higham, 1996) for more references. Although (Sterbenz, 1974) may seem dated, it is a good text. In §9.2 an algorithm for computing the average of two floating-point numbers is presented, carefully weighing the merits of the three formulae

$$(x+y)/2 \;=\; x/2+y/2 \;=\; x+(y-x)/2. \tag{23}$$

7. Conclusion and Recommendations

We have shown that there are many intricacies surrounding the use of floating-point numbers. Our examples are not new, but we have selected and adapted them to be very accessible. It is not true that numerical difficulties occur only in very special situations which lie outside the domain of pre-university education.

For completeness' sake, we mention that we have ignored two important areas of numerical mathematics, related to two additional sources of error:

data uncertainty the error already present in the input values, e.g., when they were obtained by physical measurement;

truncation error the error introduced by applying an inexact algorithm (that is, an algorithm which is known to produce an incorrect answer when run on an ideal machine), with the purpose of obtaining accurate answers in less time, e.g., by chopping off an infinite series or approximating a function by a polynomial.

Our recommendations can be summarized as follows:

1. *Avoid the use of floating-point numbers in computing whenever possible.*

   To teachers: When designing programming problems, there are plenty of possibilities without floating-point numbers. In fact, it is a good attitude to *forbid* your students to use floating-point numbers in their programs, unless specifically required by the assignment, because it is so hard to reason about floating-point programs.

To students: When solving programming problems whose description does not involve floating-point numbers, resist the temptation to use them.

2. *If you do want to use floating-point numbers, study the literature.*

To teachers: When setting a programming problem involving floating-point numbers, make sure that the constraints are expressed carefully and that the problem is solvable for all inputs satisfying the constraints. In particular, you need to provide evidence that your own solution is acceptable. Make sure that you avoid ill-conditioned problems.

To students: Before resorting to floating-point numbers, convince yourself that this is really necessary. Then, convince yourself that your program satisfies all constraints. In particular, check that you have not fallen into one of the 'standard' traps giving rise to an unstable algorithm.

In both cases, some form of error analysis is needed.

These recommendations apply also to competitions: just read 'organizers' for 'teachers' and 'competitors' for 'students'. However, in view of the complications brought by floating-point arithmetic, we believe that competitions can best steer clear of floating-point.

The quote from (Knuth, 1997) that we gave in Section 1 continues as follows:

> "Many serious mathematicians have attempted to analyze a sequence of floating point operations rigorously, but have found the task so formidable that they have tried to be content with plausibility arguments instead."

Computations involving floating-point numbers require careful arguments to convince oneself and – more importantly – others of the reliability of the results. The design and analysis of numerical algorithms has become a specialism of its own. It is an interesting but difficult field, well beyond the pre-university mathematics curriculum.

References

BOI2002 (2003). Triangles, competition task on Day 1. *Baltic Olympiad in Informatics*, Lithuania. `http://aldona.mii.lt/pms/olimp/english/boi2002/`

Goddijn, A. (2002). Nee is meestal Nee, maar Ja niet altijd Ja. *Pythagoras*, **(**41) 6, 25–29 (in Dutch).

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. Appendix D in *Numerical Computation Guide*. Sun Microsystems, Revision A, May 2000; originally published in *ACM Computing Surveys*. `http://docs.sun.com/source/806-3568/ncg_goldberg.html`

Graham, R. Two sums of nine square roots of integers agreeing up to 40 digits. `http://www.cs.nyu.edu/exact/realexpr/Graham.html` (accessed March 2003).

Higham, N.J. (1996). *Accuracy and Stability of Numerical Algorithms*. SIAM.

Horváth, G., and T. Verhoeff (2002). Finding the median under IOI conditions. *Informatics in Education*, **1** (1), 73–92.

IEEE (1985). *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*. IEEE.

IOI (2003). *International Olympiad in Informatics*, Internet WWW-site, URL: `http://www.IOInformatics.org/`

IOI2002 (2002). Bus Terminals, competition task on Day 2. *International Olympiad in Informatics*, Korea. `http://olympiads.win.tue.nl/ioi/ioi2002/contest/day2/bus/`

Kahan, W. (1981). *Why do we need a floating-point arithmetic standard?* Unpublished note, February 1981. `http://www.cs.berkeley.edu/ dbindel/class/cs279/why-ieee.pdf`

Kahan, W. (1996). *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic.* May 1996. `http://www.cs.berkeley.edu/ wkahan/ieee754status/IEEE754.PDF`

Knuth, D.E. (1997). *The Art of Computer Programming: Seminumerical Algorithms, vol. 2*. 3rd Ed. Addison-Wesley.

O'Rourke, J. (1981). Advanced Problem 6369. *Amer. Math. Monthly*, **88** (10), 769. Also known as Problem 33 in the *Open Problems Project*. `http://cs.smith.edu/ orourke/TOPP/` (Accessed March 2003).

Overton, M.L. (2001). *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM.

Press, W.H. *et al.* (1989). *Numerical Recipes in Pascal: The Art of Scientific Computing*. Cambridge University Press, 1989. `http://www.nr.com/`

Sterbenz, P.H. (1974). *Floating-Point Computation*. Prentice-Hall.

G. Horváth is an associate professor in computing science at the University of Szeged, Hungary. He obtained his PhD (Functor State Machines, 1981) from the University of Szeged. His specialty is algorithms and data structures. He chaired the Scientific Committee of IOI 1996 in Hungary and contributed tasks to various competitions. From 1999 to 2001 he was a member of the IOI Scientific Committee.

T. Verhoeff is an assistant professor in computing science at Eindhoven University of Technology (TUE), the Netherlands. He obtained his PhD (A Theory of Delay-Insensitive Systems, 1994) from TUE. His current reseach area is software construction. He chaired the Scientific Committee of IOI 1995 in The Netherlands and contributed tasks to various competitions. In 1999, he was Finals Directors of the ACM ICPC World Finals held in Eindhoven, The Netherlands. He chairs the IOI Scientific Committee since 1999.

Skaitinių uždavinių mokant informatikos ikiuniversitetinėse studijose ir varžybose problemos

Gyula HORVÁTH, Tom VERHOEFF

Dažnai nepakankamai įvertinami sunkumai, kylantys vartojant slankiojo kablelio skaičius programavime. Tai atsitinka ikiuniversitetinėse studijose bei varžybose, kur tikima, kad slankiojo kablelio aritmetika yra geras realiųjų skaičių aproksimavimas. Deja, dauguma realiųjų skaičių operacijų taisyklių netinka slankiojo kablelio skaičiams. Straipsnyje nagrinėjamas šis neatitikimas, pateikiami keturi jį iliustruojantys pavyzdžiai. Aprašomos šios sąvokos: slankiojo kablelio skaičių sistemos struktūra naudojant tikslumą ir rėžius; netikslumai atliekant slankiojo kablelio skaičių operacijas; klaidos skleidimas skaitiniuose algoritmuose ir to nustatymas analizuojant klaidas; slankaus kablelio dvejetainės formos standartinė išraiška; prastinimas; absoliutinių bei santykinių paklaidų įvertinimas; skaitinių algoritmų stabilumas; nekorektiškų sąlygų algoritmai; perpildymas. Mokant informatikos, taip pat varžybose, studentams turi būti suteikiamos žinios, formuluojamos programavimo situacijos, įvertinamos pateiktos programos bei apdorojamos variacijos, kylančios esant skirtingoms platformoms. Straipsnyje parodoma, kad slankiojo kablelio skaičių vartojimas sukelia sudėtingas problemas visose šiose srityse. Norint įveikti šiuos sunkumus reikia, kad mokytojai bei studentai įgytų patirties dirbant su skaičių teorija. Autoriai griežtai rekomenduoja vengti, kiek tai yra įmanoma, slankiojo kablelio skaičių naudojimą ikiuniversitetiniame mokyme bei varžybose. Jei vis dėlto norima naudoti slankiojo kablelio skaičius, reikia gerai įsigilinti į skaičių teoriją ir būti pasirengus atlikti klaidų analizę.