Lecture 1
INTRODUCTION

This lecture gives an orientation on the some of the issues that concern us. What is the nonrobustness phenomenon? Why does it appear so intractable, as is often remarked? Of course, the *prima facie* reason for nonrobustness is numerical errors. But this is not the full story: the key lies in understanding the geometry which is founded upon the numerical data and computation.

§1. Numerical Nonrobustness

Nonrobustness of computer systems means different things to different communities. In these notes, we are only interested in the nonrobustness that can be traced to a numerical source. In contrast, sometimes a computer solution is described as "nonrobust" if it is non-scalable (i.e., as the problem size gets larger, the solution becomes no longer practical) or "non-extendible" (i.e., when we add some simple variations, the solution breaks down). For brevity, we simply say "(non)robustness" instead of "numerical (non)robustness" throughout these notes.

The nonrobustness phenomenon is well-known and widespread, and is based on the fact that most numerical quantities in computers are approximations. Hence there is **error** which may be defined to be the difference between the exact[1] value and the computed value. Most of the time, such errors are **benign**, provided we understand that they are there and we account for them appropriately. The phenomenon that we call nonrobustness arises when benign errors lead to **catastrophic errors**. The computer system or program crashes as a result. There is no graceful recovery from such errors. Why is this? When does this occur?

Example 1 (Intersecting Lines) A simple illustration of catastrophic error is the following. Assume we are writing a simple geometry program in which we have primitives to define points and lines in the Euclidean plane. We also have primitives to intersect two lines and also test if a point is on a line. Mathematically, a point p is a pair (x, y) of real numbers, and a line ℓ is given by an equation

$$\ell : ax + by + c = 0$$

where (a, b, c) are the real parameters defining ℓ. We write $\ell = \textsc{Line}(a, b, c)$ to indicated that the line ℓ is defined by (a, b, c). Similar, $p = \textsc{Point}(x, y)$ is a definition of the point p. If $\ell' = \textsc{Line}(a', b', c')$ is another line, then the intersection of ℓ and ℓ' is the $\textsc{Point}(x, y)$ that satisfies the linear system

$$\begin{bmatrix} a & b \\ a' & b' \end{bmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = -\begin{pmatrix} c \\ c' \end{pmatrix}, \tag{1}$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\Delta} \begin{bmatrix} -b' & b \\ a' & -a \end{bmatrix} \begin{pmatrix} c \\ c' \end{pmatrix} \tag{2}$$

where $\Delta = ab' - a'b$ is the determinant. In two exceptional cases, the intersection is not a point: (i) when $\ell = \ell'$ (so there are infinitely many points of intersection) or (ii) when $\ell \| \ell'$ (the lines are parallel). In these cases, we assume that the intersection is an "undefined value" (denoted $\uparrow$). So we define the **operation** $\textsc{Intersect}(\ell, \ell')$ to return the intersection point (possibly $\uparrow$) of lines ℓ and ℓ'. This operation is easily implemented in any conventional programming language using the formula (1). Similarly, it is easy to

[1] In the present discussion, we assume that each quantity has a theoretically "exact" value. There is a more general setting in which values are inexact but quantifiably so. We will return to this later. However, the nonrobustness issue is already non-trivial in the exact setting.

program the **predicate** OnLine(p, ℓ) which returns TRUE iff p lies on line ℓ, otherwise returns FALSE. Note that $p = \text{Point}(x_0, y_0)$ lies on $\ell = \text{Line}(a, b, c)$ iff $ax_0 + by_0 + c_0 = 0$.

General Remark: All our geometric primitives (operations or predicates or constructors) are capable of returning an undefined ($\uparrow$) value. In particular, if any of its argument is undefined then the result is always undefined.

Consider the following expression:

$$\text{OnLine}(\text{Intersect}(\ell, \ell'), \ell) \tag{3}$$

Mathematically, the value of this expression is TRUE unless Intersect(ℓ, ℓ') is undefined. Yet, any simple implementation of these primitives on a computer will fail this property. That is to say that for "most choices" of ℓ and ℓ', the expression will not evaluate to TRUE. The reason for this is that clear – computer arithmetic has fixed precision and approximate. The expression we want to evaluate is too sensitive to even the slightest error for most inputs. An exercise below will attempt to quantify "most choices" in the above remark. ■

Operations such as line intersection are very basic, and they form the basis of more complex operations in geometry libraries. If such basic operations are nonrobust, applications built on top of the libraries will face the same nonrobustness problem but in more intractable forms. It is thus no surprise that applications such as geometric modelers are highly susceptible to catastrophic errors. The property of failing to return TRUE when it should have, is a **qualitative error**, and this is a result of a **quantitative error**. Numerical errors are by nature a quantitative phenomenon, and are often benign. But under certain conditions, they become a qualitative phenomenon. We need to understand when such transitions take place, and to find measures either to detect or to avoid them.

Epsilon Tweaking. A common response of most programmers to this example is to say that the comparison "is $ax + by + c = 0$?" is too much to ask of approximate quantities. Instead, we should modify the OnLine$(\text{Point}(x, y), \text{Line}(a, b, c))$ predicate to the following

$$\text{"Is } |ax + bx + c| < \varepsilon\text{?"} \tag{4}$$

where $\varepsilon > 0$ is a small constant. This idea is completely general and widely used: everytime we need to compare something to 0, we will compare it to some small "epsilon" constant. There will in general be many different epsilons in different parts of the code. It is somewhat of an art to pick these constants properly. In the programming world, the empirical task of choosing such constants is called "epsilon tweaking".

Note that in our expression (3) above, if we make ε large enough, the predicate will surely return TRUE for all pairs of intersecting lines. But it introduces problems of its own, as we have changed the underlying geometry. This is sometimes called **epsilon geometry**. But what exactly is this geometry? One interpretation is that we are now dealing with lines or points that are "fat". A point $p = \text{Point}(x, y)$ is really a disk centered about (x, y) with radius $\varepsilon_1 \geq 0$. Similar, a line $\ell = \text{Line}(a, b, c)$ really represents a strip S of region with the mathematical line as the axis of S. The strip has some width $\varepsilon_2 \geq 0$. For simplicity, suppose $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$. Then the OnLine$(p, \ell)$ predicate is really asking if the fat point p intersects the fat line ℓ. This is illustrated in figure 1.

This interpretation of (4) is fine after we normalize the equation of our line so that $a^2 + b^2 = 1$. To normalize the coefficients (a, b, c), we simply divide each coefficient by $\sqrt{a^2 + b^2}$. The distance of any point (x, y) from the normalized $Line(a, b, c)$ is given by $|ax+by+c|$. Thus the epsilon test (4) amounts to checking if p is within distance ε from the line ℓ.

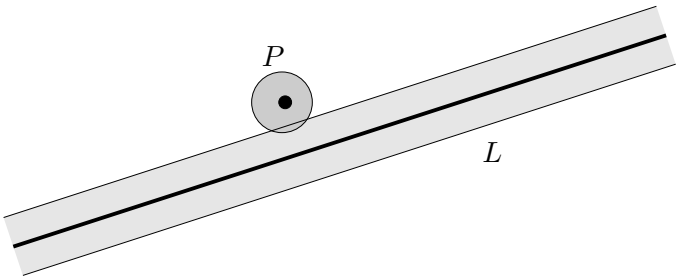

Figure 1: A fat point p intersecting a fat line ℓ.

More important, we observe that by resorting to epsilon-tweaking, we have exchanged "plain old Euclidean geometry" (POEG) for some yet-to-be-determined geometry. Even if a consistent interpretation can be obtained, it will be be pretty hard to reason correctly with such modified geometries. For instance, what is the intersection of two fat lines? Is it a fat point or it is a diamond-shaped region? Adopting such a geometry has wide-spread (mostly disastrous) consequences for computation: most of our intuitions and algorithms are guided by Euclidean geometry, and we will find all kinds of surprises in trying to design algorithms for such geometries.

Interval Geometry. There is an alternative interpretation of the test (4). Here, we assume that each point is only known up to some ball of radius ε. Then the test (4) is only useful for telling us if the point definitely does not lie on the line. But the conditions for knowing if a point is definitely on the line can only be determined with additional assumptions. We then proceed into a three-valued logic in our programs, in which "partial predicates" gives answers of yes/no/maybe. This is basically the geometric analogue of the well-known idea of **interval arithmetic**. This is not unreasonable in some applications, but a common problem is that the "maybe" answers might rapidly proliferate throughout a computation and lead to no useful results. Alternatively, we might think of the "geometric intervals" (unknown region about a point or a line) growing rapidly in an iterated computation.

Remarks:. In the literature, what we call "robustness" also goes under the name of **reliability**. An international journal[2] devoted to this area is **Reliable Computing** from Kluwer Academic Publishers. Two recent special journal issues devoted to robust geometric computation are [24] and [8]. An early paper to draw attention to robustness is Forrest [7]. Quoting Forrest, "I have yet to discover any correct general algorithm and suspects that although reliability is possible for limited cases, it is not achievable in general". He added that geometric modelers are "far from robust" and "we might question whether this care with arithmetic is really necessary, but numerous horror stories from practice testify to the need for action". Hoffman [13] discusses nonrobustness problems in geometric computation. Patrikalakis [22] surveys robustness issues in geometric modelling. Robustness as an key research topic is noted Computational Geometry Task Force report from Chazelle et al [3, 2], and also described in the ACM Strategic Directions in Computing Research Report from Tamassia et al [25]. A simple experiment in [14] using IEEE arithmetic for the intersecting line example (3) for 2500 pairs of intersecting lines yields a failure rate of 40%. Of course, even a single failure in a program could have potentially catastrophic consequence.

Exercises

Exercise 1.1: (a) Write a simple planar geometry program which handles points and lines. An object oriented programming language such as `C++` would be ideal, but not essential. Provide the operations

[2] From 1991 to 1994, it was known as "J. of Interval Computations".

to define points and lines, to intersect two lines and to test if a point is on a line. You must provide for undefined values, and must not attempt any epsilon-tweaking.

(b) Conduct a series of experiments to quantify our above assertion that the expression (3) will not evaluate to TRUE in a "large number" of cases ♢

Exercise 1.2: Verify that for a normalized line $\ell = \text{LINE}(a, b, c)$, the *signed distance* of $p = \text{POINT}(x, y)$ from is $ax + by + c$. Interpret this sign. ♢

Exercise 1.3: Let us quantify the frequency of failure of the expression (3) in a very simple situation. Assume that we implement the primitives in the straightforward way, as outlined above, and using IEEE double precision arithmetic (see next section for details). Assume that $\ell = \text{LINE}(0, 1, -1)$ and $\ell' = \text{LINE}(1, y, 0)$. Let y be a IEEE double. Estimate the percentage of times when the expression (3) fails (does not output a TRUE when it should have). ♢

End Exercises

§2. Fixed Precision Arithmetic

It is important to understand some basic properties about machine arithmetic representation, as they are ultimately the basis of almost all numerical computation. Furthermore, as machine arithmetic is highly optimized, they will turn out to be very useful in our solutions to nonrobustness. We review some of the main arithmetic systems in the literature.

Actually, we will mainly focus on the number representation in such systems. We use the term **fixed precision arithmetic** to mean any arithmetic system that operates on numbers that has a fixed budget of bits or space to represent their numbers. All modern machine arithmetic are fixed precision arithmetic. Furthermore, most common systems are based on numbers using a positional number system with a **fixed radix**, $\beta \geq 2$. Typically, the number is represented by a fixed number[3] of "values". A value is represented by fixed number s of digits in radix β. For instance, with $\beta = 2$ and $s = 32$, we have a typical binary integer with 32 bits assumed in many high level programming languages. There are two main variants of such numbers: fixed point or floating point. We will mostly focus on floating point numbers, but briefly review the other possibilities.

A **fixed point system** of numbers with parameters $m, n, \beta \in \mathbb{N}$ comprise all numbers of the form

$$\pm d_1 d_2 \ldots d_n . f_1 f_2 \ldots f_m$$

and d_i, f_j are digits in base β. Typically, $\beta = 2, 10$ or 16. Fixed point systems are not much used today, except for the special case of $m = 0$ (i.e., integer arithmetic). It is instructive to briefly look at another class of fixed-precision arithmetic, based on rational numbers. Matula and Kornerup [18] gives a detailed study of such systems. As in fixed-radix numbers, there are two basic versions: **fixed-slash** and **floating-slash** rational numbers. In the fixed-slash system, we consider the set of rational numbers of the form $\pm p/q$ where $0 \leq p, q \leq n$ for some n. The representable numbers in this system is the well-known **Farey Series** F_n of number theory. In the floating-slash system, we allow the number of bits allocated to the numerator and denominator to vary. If L bits are used to represent a number, then we need about $\lg L$ bits to indicate this allocation of bits (*i.e.*, the position of the floating slash). The paper Matula and Kornerup addresses questions of naturalness, complexity and accuracy of fixed-precision rational arithmetic. Other proposed

[3]For instance, a floating point number has two such values: the mantissa and the exponent, each with its own β and s.

number system include Hensel's p-adic numbers (e.g., [11, 17]) and continued fractions (e.g., [4, 28]). For general information about about number systems, see Knuth [16].

Henceforth, we focus on floating point systems. The reason is that this is the dominant system in scientific and engineering applications. A **floating point system** with parameters $\beta, t \in \mathbb{N}$ and $e_{\min}, e_{\max} \in \mathbb{Z}$ comprises al numbers of the form

$$r = m \times \beta^{e-t} = \frac{m}{\beta^t}\beta^e \tag{5}$$

where $m, e \in \mathbb{Z}$ with

$$|m| < \beta^t, \qquad e_{\min} \leq e \leq e_{\max}.$$

We call β the **base** and t the **significance**. Thus m can represented by a t-digit natural number in the usual β-ary notation. The number r in (5) is represented by a **mantissa-exponent** pair (m, e), and we denote this relation by writing

$$float(m, e) = r.$$

If $m \geq \beta^{t-1}$ or $m = 0$, then we say (m, e) is a **normal representation**. The factor β^{-t} in (5) is the normalization factor. Among the non-normal representations, we have an important subset known as **subnormal representation**: this is (m, e) where $|m| < \beta^t$ and $e = e_{\min}$. Numbers with subnormal representations are said to be **subnormal** in F. It is not hard to check that two distinct normal mantissa-exponent pairs represent distinct numbers: suppose $(m, e) \neq (m', e')$. If $e = e'$ then clearly
$float(m, e) \neq \text{float}(m', e')$. Otherwise, say $e > e'$ and we may assume that m, m' are positive. Then

$$\text{float}(m, e) = \frac{m}{\beta^t}\beta^e \geq \beta^{e-1} \geq \beta^{e'} > \frac{m'}{\beta^t}\beta^{e'} = \text{float}(m', e').$$

Thus, comparison of normalized numbers are easy. To compare unnormalized numbers, we first normalize them.

The case $\beta = 2$ is most common in hardware today; in this case, the leading bit of the mantissa representation is always 1 and this need not be explicitly represented.

Let $F(\beta, t, e_{\min}, e_{\max}) \subseteq \mathbb{R}$ denote the set of representable numbers in this system. We can also write (5) as

$$r = \pm\beta^e \times 0.d_1 d_2 \ldots d_t.$$

Here, $0 \leq d_i \leq \beta - 1$. If (m, e) is normal then $d_1 \neq 0$. Also, d_1, d_t are the most and least significant digits, respectively.

It is useful to consider a larger number system $F(\beta, t)$ in which we do not bound the exponents, *i.e.*,

$$F(\beta, t) = F(\beta, t, -\infty, +\infty).$$

Once we admit unbounded parameters in our floating point system notation, we can get the set of integers as the system $\mathbb{Z} = F(2, 0, 0, +\infty)$.

In the following, we fix a set $F \subseteq \mathbb{R}$ (and call this a number system). Unless otherwise stated, $F = F(\beta, t, e_{\min}, e_{\max})$ or $F = (\beta, t)$. Numbers of F are said to be representable.

Rounding. If $x \in \mathbb{R}$ then $\text{fl}(x)$, the **rounding** of x, denotes a representable value closest to x. Rounding is unique if there is a unique closest representable value. In case of ties, there are several alternatives: round up, round down, towards 0, away from 0, and towards even. It turns out that the last possibility has superior computational properties, and is the default in modern hardware. REMARKS: we sometimes use the more compact alternative notation of "$[x]$" instead of "$\text{fl}(x)$". When $F = \mathbb{Z}$, there is a special notation for $\text{fl}(x)$, namely "$\lfloor x \rceil$".

Resolution and Range. Because of the fixed precision in $F = F(\beta, t, e_{\min}, e_{\max})$, there are several related measures of the "finest resolution possible" in such systems. One is the **machine epsilon**, $\varepsilon_M := \beta^{1-t}$, which may be defined to be be the distance from 1.0 and the next representable number. More important for error analysis is the **unit roundoff**, defined as

$$\mathbf{u} := \frac{1}{2}\beta^{1-t} = \varepsilon_M/2. \tag{6}$$

Similarly, because of the fixed precision, it is important to specify the range of numbers that are reasonably approximated by F. It is not hard to see that if $y \in F$ then

$$\beta^{e_{\min}-1} \leq |y| \leq \beta^{e_{\max}}\left(1 - \beta^{-1}\right).$$

This is called the **range** of F.

One of the striking properties of F is the non-uniform distribution of its numbers. To understand this distribution, we consider the non-negative portion of the range of F, $[0, \beta^{e_{\max}})$. Subdivide this into half-open intervals of the form $I_e := [\beta^{e-1}, \beta^e)$,

$$[0, \beta^{e_{\max}}) = I_{-\infty} \cup I_{e_{\min}} \cup I_{e_{\min}+1} \cup \cdots \cup I_{e_{\max}}$$

where $I_{-\infty}$ is defined to be $[0, \beta^{e_{\min}-1})$.

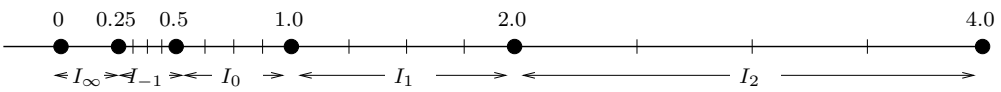

Figure 2: Non-uniform intervals in $F(2, 3, -1, 2)$: normal numbers.

In figure 2, each interval I_e (for $e > -\infty$) has exactly 3 normal numbers in its interior. In general, the other intervals I_e contains exactly $\beta^{t-1}(\beta - 1)$ representable numbers which are uniformly spaced β^{e-t+1} apart. Note that except for 0, the representable numbers in $I_{-\infty}$ are precisely the subnormal numbers. It is easy to see that if $x, x' \in I_e$ are two consecutive representable numbers then

$$\frac{|x|}{\beta^t} \leq |x - x'| \leq \frac{|x|}{\beta^{t-1}}.$$

Theorem 1 *If $x \in \mathbb{R}$ lies in the range of F, then*

$$fl(x) = x(1 + \delta), \qquad |\delta| < u$$

and also

$$fl(x) = \frac{x}{1 + \delta'}, \qquad |\delta'| \leq u.$$

Example 2 (Some Floating Point Systems)
IEEE single precision numbers refers to the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 24, -125, 128)$$

where $u \approx 5.96 \times 10^{-8}$ and range is $10^{\pm 38}$. The IEEE double precision numbers refers to the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 53, -1021, 1024)$$

where $u \approx 1.11 \times 10^{-16}$ and range is $10^{\pm 308}$. The number of bits used to represent these two systems are 32 and 64 bits, respectively. In the double format, 53 bits are dedicated to the mantissa and 11 bits for the exponent. One of the 53 bits are used for sign; but we still have 53 (not 52) bits of precision because the leading bit of the mantissa is not explicitly represented, and be assumed to be 1 for normalized numbers. The 11 bits for the exponent represent an integer e between 0 and 2048, and this number represents the exponent $e - 1023$. For error analysis, we need to know that their unit roundoffs:

$$\mathbf{u} = 2^{-24}(\text{single});\ \mathbf{u} = 2^{-53}(\text{double}).) \tag{7}$$

■

Floating Point Arithmetic. Let $\circ$ be any basic floating point operation (usually, this refers to the 4 arithmetic operations, and sometimes square-root is included). Let $\circ'$ be the corresponding operation for the numbers in F. The fundamental property of fixed precision floating point arithmetic is this: if $x, y \in F$ then

$$x \circ' y = \text{fl}(x \circ y).$$

This assumes $\circ$ is binary, but the same principle applies for unary operations. Because of this, we have the following property

$$x \circ' y = (x \circ y)(1 + \delta), \qquad |\delta| \leq u. \tag{8}$$

Any model of machine arithmetic that satisfies (8) is called a **standard model** (with unit roundoff u). Recall that the unit roundoff is usually written in bold, $\mathbf{u}$.

Note that if $x \circ y \in F$, we would hope that $\delta = 0$, but this is not required by (8).

We refer to [16, Chapter 4] for the algorithms for arithmetic in floating point systems (and about general number systems and their history). Without delving into the details, there is a generic 3-step process for performing floating point arithmetic: Suppose $F = F(\beta, t)$ for simplicity, and we want to perform the binary operation $x \circ' y$ in F,

1. (Scale) First "scale" the operands so that they share a common exponent. We normally scale the operand with the larger magnitude to match the exponent of the operand with smaller magnitude. The scaled number may no longer be normal.
2. (Operation) Carry out the desire operation $\circ$ on the two mantissas. Because of scaling, this is now essentially integer arithmetic (which is regarded as fast).
3. (Truncation) Finally, we truncate the result to t digits of precision, and normalize the result.

The Scaling Step is the key to understanding errors in floating point arithmetic: after we scale the operand with smaller magnitude, its mantissa may require much more than t digits. So, immediately after scaling, all hardware implementation will perform a truncation of the scaled operand. But truncation to what precision? We might guess that truncating to t digits is sufficient (after all the final result will only have t digits). This is almost right except for one tiny caution: in the case of addition or subtraction, we should truncate to $t+1$ digits. This extra digit is called the **guard digit**. Without this, the hardware will fail to deliver a standard model (8). This was a well-known "bug" in hardware before the IEEE standard (Exercise).

In general, because of scaling and truncation, if we add a very large number to a very small number, the effects of the smaller number may be lost. Many techniques or algorithms in floating point computation try to reorder a sequence of arithmetic operations so as to minimize such loss of precision.

§3. The IEEE Standard

In these notes, I will use the term **FP computation** to mean computing with *fixed precision* and with *floating point numbers*. We first note some negative properties of FP computation as compared to fixed point computation:

(1) Algorithms for FP arithmetic is much more complicated.

(2) The spacing between consecutive representable numbers is non-uniform, in contrast to fixed-point numbers.

(3) Error analysis for FP computation is much harder to understand.

Item (1) is an issue for hardware designers. This led the early computer designers (including von Neumann) to reject it as too complicated. In contrast,[4] fixed point arithmetic is relatively straightforward. Items (2) and (3) contribute to the many well-known and otherwise pitfalls in FP computation. There are many controversial issues about how to do FP computation correctly, issues that are still relevant today. Wilkinson, especially through his extensive error analysis of floating point computations, is credited with making floating point arithmetic better understood and accepted in main stream numerical computing.

Today, FP computation has become the *de facto* standard for computing in scientific and engineering applications. In view of the above problems, this triumph of FP computation may seem surprising. First, let us note that criterion (1) is no longer an critical issue. This is because the algorithms and circuit design methodology for FP arithmetic are well-understood and relatively stable. But what are the inherent advantages of FP computation? There seems to be mainly one: for a given budget of bits, the range of numbers which can be reasonably approximated by floating point numbers is much larger than using fixed point representation. This was critical in allowing scientific computations in many domains to proceed (in the days of slower and clumsier hardware, this spell the difference between being able to complete one computation or not at all). In some sense, this advantage is less importance with advancing computer speed and increasing hardware complexity. We will return to this remark later.

Early computers would implement floating point arithmetic in software. Alternatively, one can have an optional piece of hardware ("co-processor") to do floating point computation. Today, floating point processing is so well-established that it is considered standard hardware equipment.

The IEEE Floating-Point Standard. This usually refers to the IEEE Standard 754 [26, 9] for Binary Floating-Point Arithmetic. There is a generalization called the IEEE Standard 854 (1987) which applies to any base and to any word length. It is important to understand some basic properties of this standard because all modern computer arithmetic subscribes to it. This standard is precipitated by a growing problem in numerical computation in the 1980s. As FP computation grew in importance, hardware implementations of floating point arithmetic began to proliferate. The divergence among these architectures caused confusion in the computing community, and numerical software became largely non-portable across platforms. In 1985, the IEEE established the said standard for hardware designers. See the book of Patterson and Hennessy

[4] A comparison of the physical sizes of computer chips for FP arithmetic and for integer arithmetic will show the vast gap in their relative complexity.

[23] for some of this history. Without diminishing the importance of this standard, we should properly understand the significance of this standard vis-a-vis our robustness goal.

- It ensures consistent performance across platforms. This is no mean achievement considering the confusion preceding its introduction and acceptance. To emphasize what it *does not* mean, we state this in the negative form: *the IEEE standard ensures that if a program fails on one machine, it will also fail on another platform.*
- Because of its rational design, it can reduce the frequency of nonrobust behavior. Again, we must emphasize what it does mean: *it does not eliminate nonrobustness.*
- There are issues beyond the current standard that remain to be addressed. A key problem here is high-level language support for this standard (e.g., [6]). The IEEE Standard is usually viewed as a hardware standard. Most programming languages still do not support the IEEE standard in a systematic way, or consistently across platforms.

The IEEE standard provides for the following

- number formats (including subnormals). We already mentioned the single and double formats above.
- conversions between number formats
- special values (NaN, $\pm\infty$, ± 0)
- unambiguous results of basic operations $(+, -, \times, \div)$ as well as more advanced ones (remainder and $\sqrt{}$). Transcendental functions are not in the standard per se.
- unambiguous results for comparisons (NaN is incomparable, $+0 = -0$, etc)
- 4 rounding modes: to even (the default), up, down, to zero. [Note: a variant of rounding to zero is truncation.]
- exceptions and their handling: invalid result, overflow, divide by 0, underflow, inexact result

Exercises

Exercise 3.1: Determine all the numbers in $F = F(2, 3, -1, 2)$. What are the subnormal numbers? What is $\mathbf{u}$ and the range of F? ♢

Exercise 3.2: Prove theorem 1 about the roundoff unit $\mathbf{u}$. ♢

Exercise 3.3: Consider arithmetic in the system $F(\beta, t)$.
(i) Show that the standard model (8) holds for multiplication or division when the scaled operand is truncated to t digits (just before performing the actual operation).
(ii) Show that the standard model (8) fails for addition or subtraction when we truncate to t digits.
(iii) Give the worst case error bound in (ii).
(iv) Show that the standard model holds for addition and subtraction if we have a guard digit. ♢

End Exercises

§4. Error Analysis in FP Computation

Most numerical computing are based on the finite precision numbers, and almost invariably, using the floating point representation. Within this framework, rounding errors and consequent "instability" in algorithms are inevitable. Hence a major research activity in this setting involves the analysis of errors and the design of efficient, stable algorithms.

Within any finite precision system of numbers, there is an minimal quanta of unknown in every computed result. For roundoff analysis, this quantity is is denoted u in the previous section.

Let $\varepsilon(\widetilde{y}, y)$ denote some error measure for the quantity $\widetilde{y}$ as an approximation of y. There are two main kinds of error measures: absolute or relative. In absolute error,

$$\varepsilon_{abs}(\widetilde{y}, y) = |\widetilde{y} - y|.$$

In relative error, we have

$$\varepsilon_{rel}(\widetilde{y}, y) = \frac{|\widetilde{y} - y|}{|y|}.$$

This is undefined if $y = 0$.

In error analysis, we recognized two kinds of algorithmic errors: backward and forward. Suppose we computed an approximate value $\widetilde{y}$ to $y = f(x)$. How shall we measure the quality of $\widetilde{y}$? The error $\varepsilon(\widetilde{y}, y)$ is called the **forward error**. Alternatively, if $\widetilde{y} = f(\widetilde{x})$ for some $\widetilde{x}$, then the error $\varepsilon(\widetilde{x}, x)$ is called the **backward error**.

The concept of "stability" is often an informal notion. Higham [12, 8] declines a general definition, but says that an algorithm for $y = f(x)$ is called "backward stable" if the backward error $\varepsilon(\widetilde{x}, x)$ is "small" in some context-specific sense. There is also a notion of "mixed forward-backward stable" if both $\varepsilon(\widetilde{x}, x)$ and $\varepsilon(\widetilde{y}, y)$ are "small". In lecture 3, we treat this issues again.

The stability of an algorithm is in some sense limited by the inherent stability of the problem, namely, the sensitivity of the output to the input data. Suppose $\widetilde{y} = f(x + \Delta x)$ and f is twice differentiable. Then

$$\widetilde{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \frac{f''(x + \theta\Delta x)}{2!}(\Delta x)^2$$

$0 < \theta < 1$. Hence

$$\frac{\widetilde{y} - y}{y} = \left(\frac{x f'(x)}{f(x)}\right)\frac{\Delta x}{x} + O((\Delta x)^2)$$

We define

$$c(x) := \left|\frac{x f'(x)}{f(x)}\right|$$

as the condition number of f at x.

For example, if $f(x) = \log x$, then $c(x) = |1/\log x|$, which is large when x is close to 1.

A rule of thumb connecting these concepts is that

$$\text{forward error} \prec \text{condition number} \times \text{backward number}$$

Other concepts: significant places, cancellation, overflow and underflow.

§5. Arbitrary Precision Computation

In contrast to fixed precision arithmetic, there is an alternative in which numbers have no á priori bound on its precision (except for physical limits on computer memory, etc). We will call[5] this **arbitrary precision computation** (or "AP computation") Arbitrary precision computation is becoming more mainstream, but it still represent a relatively small fraction of all scientific computation. Unlike the ubiquitous hardware support for FP computation, AP computation is only available through software. It is the basis for the field of computer algebra, computational number theory and cryptographic computations. Through the latter applications, AP computation is more central to electronic commerce and to the growth of the internet phenomenon in general. In our quest for robustness, AP computation is one of the first steps.

Computer numbers with arbitrary precision are called **Big Numbers** and the software libraries for manipulating and performing arithmetic with such numbers are usually called **Big Number Packages**. Today, some form of Big Number packages are available in most high level programming languages. Indeed, Big Number packages are part of the standard library in the popular language `Java`. A survey of big number packages is given in [29].

There are many kinds of Big Numbers. The simplest (and the basis for all the other Big Numbers) is the **Big Integer**. An obvious way to represent a Big Integer is an array or a linked list of computer words. If each word is L bits, we can view the list as a number is base 2^L (typically, $L = 32$). The four arithmetic operations on Big Integers is easily reduced to machine arithmetic on these words. The next kind of Big Number is the **Big Rational**. Basically, a Big Rational number is represented by a pair of Big Integers (m, n) representing the rational number m/n. Let us write $m : n$ to suggest that it is the ratio that is represented. The rules for reducing rational number computation to integer computation is straightforward: thus $m : n$ and $m' : n'$ are equal iff $mn' = m'n$. Big Integers are usually represented internally in some kind of positional representation (so a Big Integer is represented by a sequence of digits $a_1 a_2 \cdots a_n$ where $0 \leq a_i < \beta$ for some natural number $\beta > 1$).

Another type of Big Number is the **Big Float**: a Big Float number can be viewed as a pair of Big Integers (m, e) representing the floating point number

$$m\beta^e \tag{9}$$

(where m is called the **mantissa** and e the **exponent** and β is the implicit base). In practice, it may be unnecessary to represent e by a Big Integer; a machine long integer may be sufficient. Note that viewing (m, e) as the number (9) is at variance with the system $F(\beta, t)$ introduced in Lecture 2. It we want an analogous system, we can can proceed as follows: assuming that m is also represented in base β, let us define

$$\ell(m) \leftarrow \left\lceil \log_\beta(|m|) \right\rceil .$$

Then we can alternatively view the pair (m, e) as the number

$$Float(m, e) := \beta^{e-\ell(m)}. \tag{10}$$

Call this the **normalized Big Float** value. Since $\beta^{-1} < m\beta^{-\ell(m)} \leq 1$, this means the normalized value lies in the interval $(\beta^{e-1}, \beta^e]$. The advantage of normalization is that we can now compare two Big Floats (of the same sign) by first comparing their exponents.

Discussion: Rational versus Integers. Conceptually, Big Rationals does not appear to be so different from Big Integers. It is less well-known that from a complexity viewpoint, Big Rational computations are

[5] Alternatively, "multi-precision computation" (or "any precision"). However, we avoid the term "infinite precision" as it could be misleading.

usually very expensive relative to integer arithmetic, which serves as our base line. One problem with Big Rational numbers $p : q$ is its non-uniqueness. To get a unique representation, we can compute the greatest common denominator $m = GCD(p, q)$ and reduce $p : q$ to $p' : q'$ where $p' = p/m$ and $q' = q/m$. Without such reductions, the numbers can quickly grow very large. This is seen in the Euclidean algorithm for computing the GCD of two integer polynomials, where the phenomenon is called intermediate expression swell. Karasick et al [15] reported that a straightforward rational implementation of determinants can cause a slow down of 5 orders of magnitude. Fortunately, there are several ways to reduce this complexity.

On the other hand, Big Float arithmetic seem to be considerably faster than Big Rational arithmetic. Basically: *Big Floats recovers some of the speed advantages of Big Integer arithmetic while retaining the ability of Big Rationals to provide a dense approximation of the real numbers.* We cannot always replace Big Rationals by Big Floats since the latter represent a proper subset of the rational numbers. However, if approximate arithmetic can be used, the advantages of Big Float should be exploited. This is discussed in Lecture 4.

Interval Arithmetic. Another aspect that can be introduced into Big Number packages is interval arithmetic. Basically, **interval arithmetic** [20] amounts to replacing each number x by an interval $[a, b]$ that contains x. The obvious representation of intervals is the **end-point representation**, *i.e.*, $[a, b]$ is represented by the pair (a, b) of its end-points. But we can also have the **center-point representation** in which a pair (x, u) of numbers represents the interval $[x - u, x + u]$. Think of u as the **uncertainty** about the true value of x. What have we gained by this? If the desired interval is small, then u may need only a few bits to represent. Then the pair (x, u) may need much fewer bits than the representation by an interval. The Big Float package implemented in our `Real/Expr` and `Core Library` [29] uses this representation. Here, x is a Big Float (so it is really a mantissa-exponent pair) but u is restricted to on computer word for efficiency.

This leads us to the concept of **significance arithmetic**. First let us clarify the notion of "significance". Assuming a normalized Big Float system (10), we say that the big float number (m, e) has $\ell(m)$ significant digits. If we now introduce an uncertainty $u \geq 0$ into this representation, the triple (m, e, u) is now represents the interval

$$Float(m, e, u) := [Float(m - u, e), Float(m + u, e)].$$

In this case, the significance of (m, e, u) is defined to be $\ell(m) - \ell(u)$. In other words, we can trust only the first significance number digits of (m, e, u). Significant arithmetic, like interval arithmetic, attempt to only retain the significance warranted by the input significance. Significance arithmetic is usually traced to the work of Metropolis [1, 19]. An interesting variation of significance arithmetic arose in the `Real/Expr` package with its concept of mixed precision (which combines relative and absolute precisions). The thesis of Ouchi [21] derives the algorithms and error bounds for the basic arithmetic operations as well as square root extraction.

Survey of Big Number Packages. We summarize the survey of Dube and Yap [].

P-Adic Numbers. Another approach to exact arbitrary precision arithmetic can be based on Hensel's p-adic numbers [10, 11, 5].

Exercises

Exercise 5.1: The area of a triangle with side-lengths of a, b, c is given by a formula

$$\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \qquad s = (a + b + c)/2.$$

This formula was derived in Book I of *Metrica*, by Heron who lived in Alexandria, Egypt, from approximately 10 to 75 A.D.. If the triangle is needle-like (say, c is very small compared to a, b) the straightforward evaluation of this formula using machine arithmetic can be very inaccurate.
(i) Give a straight forward `C++` implementation of this formula. Use the input data found in the first three columns of the following table:

Problem	a	b	c	Naive	Kahan's
1	10	10	10	43.30127019	43.30127020
2	−3	5	2	**2.905**	Error
3	100000	99999.99979	0.00029	**17.6**	9.999999990
4	100000	100000	1.00005	500**10.0**	50002.50003
5	99999.99996	99999.99994	0.00003	**Error**	1.118033988
6	99999.99996	0.00003	99999.99994	**Error**	1.118033988
7	10000	50000.000001	15000	**0**	612.372435**8**
8	99999.99999	99999.99999	200000	**0**	Error
9	5278.64055	94721.35941	99999.99996	**Error**	0
10	100002	100002	200004	0	0
11	31622.77662	0.000023	31622.77661	0.**447**	0.327490458
12	31622.77662	0.0155555	31622.77661	246.**18**	245.9540000

Values in bold font indicate substantial error. Your results should be comparable to the results in the first 4th column.
(ii) Now implement Kahan's prescription: first sort the lengths so that $a \geq b \geq c$. If $c - (a - b) < 0$ then the data is invalid. Otherwise use the following formula

$$\Delta = \frac{1}{4}\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}.$$

In these formulas, the parethesis are significant. Compare your results with the 5th column of the above table.
(iii) Convert your programs in both parts (i) and (ii) into `CORE` programs and run at level III, with guaranteed precision of 64 bits. What conclusions do you draw?
NOTES: An extensive discussion of the machine arithmetic computation of this formulation is found in W. Kahan's paper "Miscalculating Area and Angles of a Needle-like Triangle" (work in progress), July 19, 1999. This paper may be downloaded from `http://cs.berkeley.edu/~wkahan/Triangle.pdf`. The above table is an excerpt from this paper. ♢

End Exercises

§6. Why Robustness is Important

While most people acknowledge the importance of numerical robustness, few seem demand it. One reason is the implicit idea that just being able to achieve a certain computation correctly *most* of the time is enough of an achievement. Another reason is that 100 % robustness is thought impossible to achieve. The premise of these lecture notes is that these assumptions can be challenged in many situations. In any case, our main goal is to create the tools and technology to make robust computation relatively fast and easy to use.

Mission Critical Applications. The importance of numerical robustness in critical missions is obvious. There are now many examples where numerical errors led to disastrous failures. One example comes from the Patriot Missile defense system during the 1990 Gulf War [12]. In 1996, the Ariane 5 rocket exploded on

a maiden flight; an inquiry[6] revealed that a numerical error and bad exception handling was the source of the trouble.

Programmer Productivity. The fact that programmers and scientists spend considerable amount of their time debugging numerical errors is a serious drag on productivity. In many of these situations, current technology is able to remove these issues.

Barrier to Full Automation. The lack of fully robust software is often the chief barrier to complete automation of a process. An example is in mesh-generation. There is good evidence that this problem should be largely routine in many applications. Yet no fully robust mesh generation system is currently known. As a result, engineers and computational scientists must be ready to manually tweak their inputs to the mesh generator to prevent it from crashing. Alternatively, they must manually fix the inconsistent output from these generators. Since mesh generation is a critical step before many simulations and computations, there is great economic motivation for solving this problem once and for all.

§7. Common Misconceptions

Exact computation is impossible. It is often said in textbooks that computer arithmetic is inherently inexact, and we must live with numerical errors. There are certain computations that cannot be handled exactly. But often these comments are directed to examples such as computing the square root of 2. This is impossible to represent in the common number representation, but there are other ways to represent $\sqrt{2}$ precisely, and for which there are methods for performing arithmetic and other common operations.

Nonrobustness is unimportant because it is a rare event. It is true that, in any given software, the inputs that causes nonrobust behavior are in a certain precise sense "rare". The problem is that these rare events do occur at a non-neglible frequency. Further, when they occur, we often have no recourse except to abort the computation.

There are situations where aborting a computation is an acceptable situation. For instance, consider a software for creating meshes for a geometric model. It turns out that many meshing software is nonrobust. Practitioners get around this by perturbing their models. For instance, by experimentally changing the coordinate system of the geometric model, we can usually avoid crashing the software.

[6]The following quote is taken from Kahan and Darcy (see `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`). "A commission of inquiry with perfect hindsight blamed the disaster upon inadequate testing of the rocket s software. What software failure could not be blamed upon inadequate testing ? The disaster can be blamed just as well upon a programming language (Ada) that disregarded the default exception-handling specifications in IEEE Standard 754 for Binary Floating-Point Arithmetic. Here is why: Upon launch, sensors reported acceleration so strong that it caused Conversion-to-Integer Overflow in software intended for recalibration of the rocket s inertial guidance while on the launching pad. This software could have been disabled upon rocket ignition but leaving it enabled had mistakenly been deemed harmless. Lacking a handler for its unanticipated overflow trap, this software trapped to a system diagnostic that dumped its debugging data into an area of memory in use at the time by the programs guiding the rocket s motors. At the same time control was switched to a backup computer, but it had the same data. This was misinterpreted as necessitating strong corrective action: the rocket s motors swivelled to the limits of their mountings. Disaster ensued. Had overflow merely obeyed the IEEE 754 default policy, the recalibration software would have raised a flag and delivered an invalid result both to be ignored by the motor guidance programs, and the Ariane 5 would have pursued its intended trajectory."

Nonrobustness is a mere inconvenience, but has no economic consequence. So the status quo is that meshing software is nonrobust. It is only just a nuisance? We think it is not: the productivity of researchers is impacted. Worst, it becomes a bottleneck in the larger view of meshing. Meshing is one of the preparatory steps for numerical simulations and other modeling applications. With nonrobust meshing, the rest of the simulation or modeling process cannot be fully automated: an expert who understand the meshing geometry must be in the loop. More generally, *nonrobustness is a serious impediment to industrial automation*.

Avoid ill-conditioned inputs. It is well-known that we can lose significance very quickly with ill-conditioned inputs. So one response of numerical analysts to our predicament it to suggest that we should avoid ill-conditioned inputs. For instance, if the input is ill-conditioned, we should first perform some random perturbation. Unfortunately, we note that ill-conditioned inputs are often a deliberate feature of the input data. For example, in engineering and architectural designs, multiple collinear points is likely to be a feature rather than an accident. So a random perturbation destroys important symmetries and features in the data.

Use stable algorithms. Stability is an important concept in FP computation. The definition[7] of what this means is usually informal, but see a nice definition in [27]. Higham [12, p.30] gives guidelines for designing numerically stable algorithm. The problem is that stable algorithms only serves to reduce the incidence of non-robustness, not eliminate it.

§8. APPENDIX

In this appendix, we gather some common notations that will be used in these lectures.

The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, \ldots\}$. The other systems of numbers are integers $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$, rational numbers $\mathbb{Q} = \{n/m : n, m \in \mathbb{Z}, n \neq 0\}$, real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$.

For any positive real number x, we write $\lg x$ and $\ln x$ instead of $\log_2 x$ and $\log_e x$.

References

[1] R. L. Ashenhurst and N. Metropolis. Error estimates in computer calculation. *Amer. Math. Monthly*, 72(2):47–58, 1965.

[2] Bernard Chazelle, et al. The computational geometry impact task force report: an executive summary. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 59–66. Springer, 1996. Lecture Notes in Computer Science No. 1148.

[3] Bernard Chazelle, et al. Application challenges to computational geometry: CG impact task force report. Tr-521-96, Princeton Univ., April, 1996. See also URL cs.princeton.edu/~chazelle/.

[4] D. G. Cantor, P. H. Galyean, and H. G. Zimmer. A continued fraction algorithm for real algebraic numbers. *Math. of Computation*, 26(119):785–791, 1972.

[5] K. Dittenberger. *Hensel Codes: An Efficient Method for Exact Numerical Computation*. PhD thesis, Johannes Kepler Universitaet, Linz, 1985. Diplomarbeit, Institute for Mathematics.

[7] We are indebted to Professor Michael Overton for this remark.

[6] S. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. Ph.d. thesis, New York University, 1999.

[7] A. R. Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In D. F. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, 1987.

[8] S. Fortune. Introduction (editorial for special issue on implementation of geometric algorithms), 2000.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[10] R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error-Free Computation*. Springer-Verlag, New York, 1984.

[11] E. C. R. Hehner and R. N. S. Horspool. A new representation of the rational numbers for fast easy arithmetic. *SIAM J. Computing*, 8(2):124–134, 1979.

[12] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[13] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3), March 1989.

[14] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric libraries. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999. Download, `ftp://cs.nyu.edu/pub/local/yap/exact/core.ps.gz`.

[15] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[16] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Boston, 2nd edition edition, 1981.

[17] N. Koblitz. *P-adic numbers, p-adic analysis, and zeta-functions*. Springer-Verlag, New York, 1977.

[18] D. W. Matula and P. Kornerup. Foundations of finite precision rational arithmetic. *Computing*, Suppl.2:85–111, 1980.

[19] N. Metropolis. Methods of significance arithmetic. In D. A. H. Jacobs, editor, *The State of the Art in Numerical Analysis*, pages 179–192. Academic Press, London, 1977.

[20] R. E. Moore. *Interval Analysis*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[21] K. Ouchi. Real/Expr: Implementation of an exact computation package. Master's thesis, New York University, Department of Computer Science, Courant Institute, January 1997.

[22] N. M. Patrikalakis, W. Cho, C.-Y. Hu, T. Maekawa, E. C. Sherbrooke, and J. Zhou. Towards robust geometric modellers, 1994 progress report. In *Proc. 1995 NSF Design and Manufacturing Grantees Conference*, pages 139–140, 1995.

[23] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[24] H. Ratschek and J. G. R. (guest editors). Introduction to the special issue: What can one learn from box-plane intersections?, 2000. Special Issue on Reliable Geometric Computations.

[25] Roberto Tamassia, et al. Strategic directions in computational geometry working group report. *ACM Computing Surveys*, 28(4), Dec. 1996.

[26] The Institute of Electrical and Electronic Engineers, Inc. IEEE Standard for binary floating-point arithmetic, 1985. ANSI/IEEE Std 754-1985.

[27] L. N. Trefethen and D. Bau. *Numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[28] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 2000. A version is available at URL `ftp:/Preliminary/cs.nyu.edu/pub/local/yap/algebra-bk`.

[29] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–486. World Scientific Press, Singapore, 1995. 2nd edition.