It is better to solve the right problem the wrong way than to solve the wrong problem the right way.

The purpose of computing is insight, not numbers.

– Richard Wesley Hamming (1915–1998)

Lecture 2
MODES OF NUMERICAL COMPUTATION

To understand numerical nonrobustness, we need to understand computer arithmetic. But there are several distinctive modes of numerical computation: symbolic mode, floating point (FP) mode, arbitrary precision mode, etc. Numbers are remarkably complex objects in computing, embodying two distinct sets of properties: quantitative properties and algebraic properties. Each mode has its distinctive number representations which reflect the properties needed for computing in that mode. Our main focus is on the FP mode that is dominant in scientific and engineering computing, and in the corresponding representation popularly known as the IEEE Standard.

§1. Diversity of Numbers

Numerical computing involves numbers. For our purposes, numbers are elements of the set $\mathbb{C}$ of complex numbers. But in each area of application, we are only interested in some subset of $\mathbb{C}$. This subset may be $\mathbb{N}$ as in number theory and cryptographic applications. The subset may be $\mathbb{R}$ as in most scientific and engineering computations. In algebraic computations, the subset might be $\mathbb{Q}$ or its algebraic closure $\overline{\mathbb{Q}}$.

These examples show that "numbers", despite their simple unity as elements of $\mathbb{C}$, can be very diverse in manifestation. Numbers have two complementary sets of properties: quantitative ("numerical") properties and algebraic properties. It is not practical to provide a single representation of numbers to cover the full range of these properties. Otherwise, computer systems might as well provide a single number type, "the" complex number type. Depending on the application, different aspects of quantitative properties or algebraic properties would be emphasized or supported. Over the years, different computing fields have developed suitable number representation to provide just the needed properties. Corresponding to these number representations, there also evolved corresponding **modes of numerical computation**. We briefly review a few of these modes:

- The **symbolic mode** is best represented by computer algebra systems such as `Macsyma`, `Maple` or `Mathematica`. In the present context of numerical computing, perhaps the most important subclass of $\mathbb{C}$ in the symbolic mode is the algebraic numbers $\overline{\mathbb{Q}}$. A simple example of an algebraic number is $\sqrt{2}$. Here, there are two common representations. A number $\alpha \in \mathbb{Q}[\beta] \subseteq \overline{\mathbb{Q}}$ can be represented by a polynomial $A(X) \in \mathbb{Q}[X]$ modulo $B(X)$ where $B(X)$ is the minimal polynomial of β. This representation is useful if we are only interested in the algebraic properties of numbers. It is sufficient to model the field operations and to check for equality. When α is real, and we are interested in the quantitative properties of numbers, then the polynomial $A(X)$ is inadequate. Instead, α can use the **isolated interval representation**, comprising of a polynomial-interval pair $(A(X), I)$ where α is the only root of $A(X)$ inside the interval I. For instance, if $\alpha = \sqrt{2}$ then we could choose $A(X) = X^2 - 2$

and $I = [1, 2]$. We can perform the arithmetic operations on such isolated interval representations. The quantitative properties of numbers are captured by the interval I, which can of course be arbitrarily narrowed. Both representations are exact.

- The unquestioned form of numerical computing in most scientific and engineering applications involves machine floating point numbers. We refer to this as the **FP mode** for "floating point" and "fixed precision", both of which are characteristic of this mode. Thus, $\sqrt{2}$ is typically represented by 64 bits in some floating point format, and converts to the printed decimal value of 1.4142135623731. In modern hardware, this format invariably conforms to the IEEE Standard [16]. The FP mode is very fast because of such hardware support. It is the "gold standard" whereby other numerical modes are measured against. One goal of numerical algorithm design in the FP mode is to achieve the highest numerical accuracy possible using machine arithmetic directly on the number representation alone (perhaps after some re-arrangement of computation steps). Over the last 50 years, numerical analysts have developed great insights into the FP mode of computation.

  The characterization "fixed precision" needs clarification since all FP algorithms can be regarded as parametrized by a precision number θ ($0 \le \theta < 1$). Most algorithms will produce answers that converge to the exact answer as $\theta \to 0$ (see Chaitin-Chatelin and Frayssé [7, p. 9]). In practice, FP algorithms are "precision oblivious" in the sense that their operations do not adapt to the θ parameter.

- The **arbitrary precision mode** is characterized by its use of Big Number types. The precision of such number types is not fixed. Because of applications such as cryptography, such number types are now fairly common. Thus the `Java` language comes with standard Big Number libraries. Other well-known libraries include the GNU Multiprecision Package `gmp`, the `MP` Multiprecision Package of Brent [5], the `MPFun` Library of Bailey [3], and `NTL` from Shoup. Surveys of Big Numbers may be found in [11, 37].

  The capabilities of Big Number packages can be extended in various ways. Algebraic roots are not normally found in Big Number packages, but the PRECISE Library [21] provides such an extension. Arbitrary precision arithmetic need not be viewed as monolithic operations, but can be performed incrementally. This gives rise to the **lazy evaluation mode** [24, 4]. The `iRRAM` Package of Müller [26] has the interesting ability to compute limits of its functions. The ability to reiterate an arbitrary precision computation can be codified into programming constructs, such as the precision begin-end blocks of the `Numerical Turing` language [15].

- The **validated mode** refers to a computational mode in which computed values are represented by intervals which contain the "true value". Properties of the exact answer can often be inferred from such intervals. For instance, if the interval does not contain 0, then we can infer the exact sign of the true value. This mode often goes by the name of **interval arithmetic**. It is orthogonal to the FP mode and arbitrary precision mode, and thus can be combined with either one. For instance, the Big Float numbers in `Real/Expr` [37] and also in PRECISE are really intervals. This amounts to automatic error tracking, or significance arithmetic.

- The **guaranteed precision mode** is increasingly used by computational geometers working on robust computation [36]. It is encoded in software libraries such as `LEDA`, `CGAL` and `Core Library`. In this mode, the user can freely specify an *á priori* precision bound for each computational variable; the associated library will then compute a numerical value that is guaranteed to to this precision. In its simplest form, one simply requires the correct sign – this amounts to specifying one relative-bit of precision [35]. Guaranteed sign computation is enough to achieve robustness for most basic geometric problems. This mode is stronger than the validated mode because the precision delivered by a validated computation is an *á posteriori* one, obtained by forward propagation of error bounds.

In this Chapter, we focus on the FP mode and briefly touch on arbitrary precision mode and validated mode. The symbolic mode will be treated in depth in a later Chapter.

§2. Floating Point Arithmetic

It is important to understand some basic properties of machine arithmetic, as this is ultimately the basis of most numerical computation, including arbitrary precision arithmetic. As machine floating-point arithmetic is highly optimized, we can exploit them in our solutions to nonrobustness; this remark will become clear when we treat filters in a later chapter. An excellent introduction to numerical floating point computation and the IEEE Standard is Overton [27].

We use the term **fixed precision arithmetic** to mean any arithmetic system that operates on numbers that have a fixed budget of bits to represent their numbers. Typically, the number is represented by a fixed number of "components". For instance, a floating point number has two such components: the mantissa and the exponent. Each component is described by a natural number. The standard representation of natural numbers uses the well-known positional number system in some radix $\beta \geq 2$. An alternative name for radix is "base". Here β is a natural number and the **digits** in radix β are elements of the set $\{0, 1, \ldots, \beta - 1\}$. A positional number in radix $\beta \geq 2$ is represented by a finite sequence of such digits. For instance, humans communicate with each other using $\beta = 10$, but most computers use $\beta = 2$. There are two main forms of machine numbers: fixed point or floating point.

Fixed Point Systems. A **fixed point system** of numbers with parameters $m, n, \beta \in \mathbb{N}$ comprises all numbers of the form

$$\pm d_1 d_2 \ldots d_n . f_1 f_2 \ldots f_m \tag{1}$$

and $d_i, f_j \in \{0, 1, \ldots, \beta - 1\}$ 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 [22] gave a study of such systems. In analogy to fixed point and floating point numbers, we also **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). Matula and Kornerup address questions of naturalness, complexity and accuracy of fixed-precision rational arithmetic in their paper. Other proposed number system include Hensel's p-adic numbers (e.g., [13, 20]) and continued fractions (e.g., [6]). Note that in p-adic numbers and continued fractions, the number of components is unbounded. For general information about about number systems, see Knuth [19].

All these alternative number systems ultimately need a representation of the natural numbers $\mathbb{N}$. Besides the standard β-ary representation of $\mathbb{N}$, we mention an alternative[1] called **β-adic numbers**. A digit in β-adic number is an element of $\{1, 2, \ldots, \beta\}$, and a sequence of such digits $d_n d_{n-1}, \ldots, d_1 d_0$ represents the number $\sum_{i=0}^{n} d_i \beta^i$. This equation is identical to the one for β-ary numbers; but since the d_i's are non-zero, every natural number has a unique β-adic representation. In particular, 0 is represented by the empty sequence. In contrast to β-adic numbers, the usual β-ary numbers are non-unique: $.1 = 0.100 = 00.1$.

Floating Point Systems. Given natural numbers $\beta \geq 2$ and $t \geq 1$, the **floating point system** $F(\beta, t)$ comprises all numbers of the form

$$r = m \times \beta^{e-t+1} = \frac{m}{\beta^{t-1}} \beta^e \tag{2}$$

where $m, e \in \mathbb{Z}$ and $|m| < \beta^t$. We call β the **base** and t the **significance** of the system. The pair (m, e) is a **representation** of the number r, where m is the **mantissa** and e the **exponent** of the representation. When exponents are restricted to lie in the range

$$e_{\min} \leq e \leq e_{\max},$$

we denote the corresponding subsystem of $F(\beta, t)$ by

$$F(\beta, t, e_{\min}, e_{\max}). \tag{3}$$

Note that $F(\beta, t)$ is essentially the system $F(\beta, t, -\infty, +\infty)$. When r is represented by (m, e), we may write

$$\text{float}(m, e) = r.$$

[1]There is a conflict in terminology here when numbers of the form $m2^n$ $(m, n \in \mathbb{Z})$ are called **dyadic numbers**. Such numbers are also known as binary floating point numbers.

For instance, $\text{float}(2^{t-1}, 0) = 1$ and $\text{float}(2^{t-1}, 2) = 4$ in $F(2, t)$.

Sometimes, e is called the **biased exponent** in (2) because we might justifiably[2] call $e - t + 1$ the "exponent". Using the biased exponent will facilitate the description of the IEEE Standard, to be discussed shortly. Another advantage of using a biased exponent is that the role of the t parameter (which controls precision) and the role of $e_{\min}, e_{\max}$ (which controls range) are clearly separated. The expression m/β^{t-1} in (2) can be written as $\pm d_1.d_2d_3 \cdots d_t$ where m is a t-digit number $m = \pm d_1d_2 \cdots d_t$ in β-ary notation. As usual, $d_i \in \{0, 1, \ldots, \beta - 1\}$ are β-ary digits. The equation (2) then becomes

$$r = \pm\beta^e \times d_1.d_2d_3 \cdots d_t = \text{float}(m, e). \tag{4}$$

In this context, d_1 and d_t are (respectively) called the **leading** and **trailing digits** of m. The number $\pm d_1.d_2d_3 \cdots d_t = m/\beta^{t-1}$ is also called the **significand** of r.

In modern computers, β is invariably 2. We might make a case for $\beta = 10$ and some older computers do use this base. In any case, all our examples will assume $\beta = 2$.

We classify the representations (m, e) into three mutually disjoint types:
(i) If $|m| \ge \beta^{t-1}$ or if $(m, e) = (0, 0)$, then (m, e) is a **normal representation**. When $m = 0$, we just have a representation of zero. Thus, the normal representation of non-zero numbers amounts to requiring $d_1 \ne 0$ in (4).
(ii) If $e = e_{\min}$ and $0 < |m| < \beta^{t-1}$ then (m, e) is a **subnormal representation**. Note that when $e_{\min} = -\infty$, there are no subnormal representations since e and m are finite values (by assumption).
(iii) All other (m, e) are **denormalized representations**.

In the following discussion, we assume some $F = F(\beta, t, e_{\min}, e_{\max})$. Numbers in F are said to be **representable** or **floats**. **Normal** and **subnormal numbers** refers to numbers with (respectively) normal and subnormal representations.

We claim that *every representable number is either normal or subnormal, but not both.* In proof, first note that the normal numbers and subnormal numbers are different: assuming $e_{\min} > -\infty$, the smallest non-zero normal number is $\text{float}(\beta^{t-1}, e_{\min}) = \beta^{e_{\min}}$, which is larger than the largest subnormal number $\text{float}(\beta^{t-1} - 1, e_{\min})$. Next, consider any denormalized representation (m, e). There are two possibilities: (a) If $m = 0$ then $float(m, e) = 0$ which is normal. (b) If $m \ne 0$ then $|m| < \beta^{t-1}$. So the leading digit of m is 0. Let $d_i = 0$ for $i = 1, \ldots, k$ and $d_{k+1} \ne 0$ for some $k \ge 1$. Consider the representation $(m\beta^\ell, e - \ell)$ where $\ell = \min\{k, e - e_{\min}\}$. It is easy to see that this is either normal ($\ell = k$) or subnormal ($\ell < k$). This shows that $\text{float}(m, e)$ is either normal or subnormal, as claimed. The transformation

$$(m, e) \longrightarrow (m\beta^\ell, e - \ell), \tag{5}$$

which is used in the above proof, is called **normalization** (even though the result might actually be subnormal).

We claim that *normal and subnormal representations are unique and they can easily be compared.* Subnormal representations are clearly unique. They are also smaller than normal numbers. To compare two subnormal representations, we just compare their mantissas. Next consider two normal representations, $(m, e) \ne (m', e')$. We may assume that $mm' > 0$ since otherwise the comparison can be based on the signs of m and m' alone. If $e = e'$ then clearly the comparison is reduced to comparing m with m'. Otherwise, say $e > e'$. If $m > 0$ then we conclude that $\text{float}(m, e) > \text{float}(m', e')$ as shown in the following:

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

If $m < 0$, we similarly conclude that $\text{float}(m, e) < \text{float}(m', e')$.

Resolution and Range. In the system $F = F(\beta, t, e_{\min}, e_{\max})$, there are two related measures of the "finest resolution possible". 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 larger representable number, i.e., $\text{float}(\beta^{t-1} + 1, 0)$. More important for error analysis is the **unit roundoff**, defined as

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

[2]It is less clear why we use "$e - t + 1$" instead of "$e - t$". Mathematically, "$e - t$" seems preferable but "$e - t + 1$" is a better fit for the IEEE standard.

We wish to define the "range of F". If y is normal then

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

Thus we define the **normal range** of F to be

$$(-\beta^{1+e_{\max}}, -\beta^{e_{\min}}] \cup [\beta^{e_{\min}}, \beta^{1+e_{\max}}).$$

The **subnormal range** is the open interval $(-\beta^{e_{\min}}, \beta^{e_{\min}})$. Note that 0 is normal but lies in the subnormal range. The **range** of F is the union of the normal and subnormal ranges of S.

A striking feature of F is the non-uniform distribution of its numbers. Informally, the numbers in F becomes more and more sparse as we move away from the origin. This non-uniformity is both a strength and weakness of F. It is a strength because the range of F is exponentially larger than could be expected from a uniformly distributed number system with the same budget of bits. It is a weakness to the extent that algorithms and error analysis based on F are harder to understand.

To understand this non-uniform distribution, we need only consider the non-negative portion of the range of F, $[0, \beta^{1+e_{\max}})$. Subdivide this into half-open intervals of the form $I_e := [\beta^e, \beta^{e+1})$ for $e \in [e_{\min}, e_{\max}]$,

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

where $\uplus$ denotes disjoint union and $I_{-\infty}$ is defined to be $[0, \beta^{e_{\min}})$. Note that except for 0, the representable numbers in $I_{-\infty}$ are precisely the subnormal numbers.

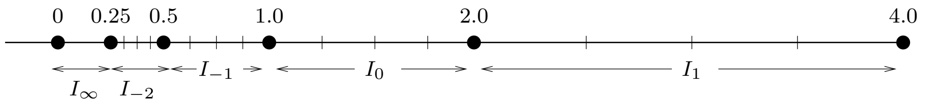

Figure 1: Non-uniform intervals in $F(2, 3, -2, 1)$: normal numbers.

For the example in Figure 1, each interval I_e (for $e \ge e_{\min}$) has exactly 4 normal numbers. In general, each interval I_e contains all the normal representations of the form (m, e), with $\beta^{t-1} \le m < \beta^t$. There are exactly $\beta^{t-1}(\beta - 1)$ numbers in I_e (this is because in (4), there are $\beta - 1$ choices for d_1 and β choices for $d_2, \ldots, d_t$). The numbers in interval I_e are uniformly spaced β^{e-t+1} apart; multiplying this by the number of normal numbers in I_e, we obtain $\beta^e(\beta - 1)$, which is the width of the interval I_e.

Rounding. It is interesting to note that the set of integers of size at most $\beta^L - 1$ is also a floating point system, $F(\beta, 1, 0, L)$. We are familiar with the concept of rounding to the nearest integer: for any real x, we can **round down** to $\lfloor x \rfloor$ (floor function) or **round up** to $\lceil x \rceil$ (ceiling function).

To discuss rounding in general, we must first generalize the ceiling and floor functions so that the role played by integers can be played by any floating point system $F = F(\beta, t, e_{\min}, e_{\max})$. Then we define $\lfloor x \rfloor$ and $\lceil x \rceil$ to be unique closest values in F that satisfy the following conditions

$$\lfloor x \rfloor \le x \le \lceil x \rceil. \tag{7}$$

To explicitly indicate that we are taking ceilings and floors relative to F, we may write $\lfloor x \rfloor_F$ and $\lceil x \rceil_F$.

There are 7 major "rounding modes". Each mode is identified by a particular **rounding function**, $\mathrm{fl} : \mathbb{R} \to F$. We have the general requirement that all rounding functions satisfy

$$\mathrm{fl}(x) \in \{\lceil x \rceil, \lfloor x \rfloor\} \subseteq F \cup \{\pm\infty\}.$$

We have already seen the two rounding functions:

$$\mathrm{fl}_1(x) = \lfloor x \rfloor \tag{8}$$

$$\mathrm{fl}_2(x) = \lceil x \rceil \tag{9}$$

Let us describe the remaining 5 rounding functions. The next two rounding modes are **round towards zero** and **round away from zero**. These corresponds to the rounding functions

$$\mathrm{fl}_3(x) = \begin{cases} \lfloor x \rfloor & \text{if } x \geq 0 \\ \lceil x \rceil & \text{if } x < 0 \end{cases}. \tag{10}$$

$$\mathrm{fl}_4(x) = \begin{cases} \lceil x \rceil & \text{if } x \geq 0 \\ \lfloor x \rfloor & \text{if } x < 0 \end{cases}. \tag{11}$$

$$\tag{12}$$

Another two rounding modes are **round to even** and **round to odd**, respectively. They depend on an additional structure of F: each number $y \in F$ is classified as even or odd, called the **parity** of y. Moreover, two consecutive numbers in F have opposite parity. By convention, the parity of 0 is even, and this uniquely fixes the parity of each number in F. This notion of parity generalizes the usual notion of even or odd integers. Now we may define the corresponding rounding functions

$$\mathrm{fl}_5(x) = \begin{cases} x & \text{if } x \in F \\ \text{the value in } \{\lfloor x \rfloor, \lceil x \rceil\} \text{ with odd parity} & \end{cases} \tag{13}$$

$$\mathrm{fl}_6(x) = \begin{cases} x & \text{if } x \in F \\ \text{the value in } \{\lfloor x \rfloor, \lceil x \rceil\} \text{ with even parity} & \end{cases} \tag{14}$$

The last rounding mode is **rounding to nearest**, with rounding function denoted $\mathrm{fl}^*(x)$ or $\lfloor x \rceil$. This is intuitively clear: we choose $\mathrm{fl}^*(x)$ to satisfy the equation

$$|\,\mathrm{fl}^*(x) - x| = \min\{x - \lfloor x \rfloor, \lceil x \rceil - x\}.$$

Unfortunately, this rule in incomplete because of the possibility of ties. So we invoke one of the other six rounding modes to break ties! Write $\mathrm{fl}_i^*(x)$ (for $i = 1, \ldots, 6$) for the rounding function where tie-breaking is determined by $\mathrm{fl}_i(x)$. Empirically, the variant fl_6^* has superior computational properties, and is the default in the IEEE standard. Hence it will be our default rule, and the notation "$\lfloor x \rceil$" will refer to this variant. Thus, $\lfloor 1.5 \rceil = 2$ and $\lfloor 2.5 \rceil = 2$. Call this the **round to nearest/even** function.

Parity Again. We give an alternative and more useful computational characterization of parity. Recall that each number in F has a unique normal or subnormal representation (m, e); we say $\mathrm{float}(m, e)$ is even iff m is even. Let us prove this notion of parity has our originally stated properties. Clearly, 0 is even by this definition. The most basic is: *there is a unique even number in the set* $\{\lfloor x \rfloor, \lceil x \rceil\}$ *when* $x \notin F$. Without loss of generality, assume $0 \leq \lfloor x \rfloor < \lceil x$ where $\lfloor x \rfloor = \mathrm{float}(m, e)$ and $\lceil x \rceil = \mathrm{float}(m', e')$. If $e = e'$ then clearly m is even iff m' is odd. If $e = e' - 1$ then $m = \beta^t - 1$ and $m' = \beta^{t-1}$. Again, m is even iff m' is odd. There are two other possibilities: $\lfloor x \rfloor = -\infty$ or $\lceil x \rceil = +\infty$. To handle them, we declare $\pm\infty$ to be even iff β is even.

Let $\mathrm{fl}(x)$ be any rounding function relative to some floating point system F. Let us prove a basic result about fl:

Theorem 1. *Assume $x \in \mathbb{R}$ lies in the range of F and $fl(x)$ is finite.*
(i) Then

$$fl(x) = x(1 + \delta), \qquad |\delta| < 2\mathbf{u} \tag{15}$$

and

$$fl(x) = \frac{x}{1 + \delta'}, \qquad |\delta'| < 2\mathbf{u}. \tag{16}$$

(ii) If $fl(x)$ is rounding to nearest, then we have

$$fl(x) = x(1 + \delta), \qquad |\delta| < \mathbf{u} \tag{17}$$

and

$$fl(x) = \frac{x}{1 + \delta'}, \qquad |\delta'| \leq \mathbf{u}. \tag{18}$$

Proof. (i) Suppose $x \in I_e$ for some e. If $x = \beta^e$ then $\mathrm{fl}(x) = x$ and the theorem is clearly true. So assume $|x| > \beta^e$. The space between consecutive representable numbers in I_e is β^{e-t+1}. Writing $\Delta = \mathrm{fl}(x) - x$, we obtain $\mathrm{fl}(x) = x + \Delta = x(1 + \Delta/x) = x(1 + \delta)$ where

$$|\delta| = \left|\frac{\Delta}{x}\right| < \frac{\beta^{e-t+1}}{|x|} < \beta^{-t+1}. \tag{19}$$

This proves (15) since $\mathbf{u} = \beta^{1-t}/2$. Note this bound holds even if $\mathrm{fl}(x)$ is the right endpoint of the interval I_e. Similarly, if $\Delta' = x - \mathrm{fl}(x)$ then $x = \mathrm{fl}(x) + \Delta' = \mathrm{fl}(x)(1 + \Delta'/\,\mathrm{fl}(x)) = x(1 + \delta')$ where

$$|\delta'| = \left|\frac{\Delta'}{\mathrm{fl}(x)}\right| < \frac{\beta^{e-t+1}}{|\,\mathrm{fl}(x)|} \le \beta^{-t+1}. \tag{20}$$

(ii) This is similar to (i) applies except that $|\Delta| \le \mathbf{u}$ (not strict inequality). The analogue of (19) is $|\delta| < \mathbf{u}$, but the analogue of (20) is $|\delta'| \le \mathbf{u}$ (not strict inequality). **Q.E.D.**

Example 1 (The IEEE Floating Point Systems).
The IEEE single precision numbers is essentially[3] the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 24, -127, 127)$$

with range $10^{\pm 38}$. The IEEE double precision numbers is essentially the system

$$F(\beta, t, e_{\min}, e_{\max}) = F(2, 53, -1023, 1023)$$

with range is $10^{\pm 308}$. Note that $e_{\min} = -e_{\max}$ in both systems. The unit roundoffs for these two systems are

$$\mathbf{u} \;=\; 2^{-24} \approx 5.96 \times 10^{-8} (\text{single}); \tag{21}$$
$$\mathbf{u} \;=\; 2^{-53} \approx 1.11 \times 10^{-16} (\text{double}). \tag{22}$$

The **formats** (i.e., the bit representation) of numbers in these two systems are quite interesting because it is carefully optimized. The number of bits used to represent the single and double precision numbers are 32 and 64 bits, respectively. In the double precision format, 53 bits are dedicated to the mantissa ("significand") and 11 bits for the exponent. In single precision, these are 24 and 8 bits, respectively. We will return later to discuss how this bit budget is used to achieve the representation of $F(2, t, e_{\min}, e_{\max})$ and other features of the IEEE standard.

Floating Point Arithmetic. Let $\circ$ be any basic floating point operation (usually, this refers to the 4 arithmetic operations, although square-root is sometimes 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

$$x \circ' y = \mathrm{fl}(x \circ y). \tag{23}$$

This assumes $\circ$ is binary, but the same principle applies for unary operations. Let us call any model of machine arithmetic that satisfies (23) the **strict model** Thus, the strict model together with Theorem 1 implies the following property

$$x \circ' y = \begin{cases} (x \circ y)(1 + \delta), & |\delta| < \mathbf{u}, \\ \frac{(x \circ y)}{(1 + \delta')}, & |\delta'| \le \mathbf{u}, \end{cases} \tag{24}$$

where $\mathbf{u}$ is the unit roundoff (see (6) and (21)). Any model of machine arithmetic that satisfies (24) is called a **standard model** (with unit roundoff $\mathbf{u}$). Note that if $x \circ y \in F$, then the strict model requires that $\delta = 0$; but this is not required by the standard model. All our error analysis will be conducted under the standard

[3]The next section will clarify why we say this correspondence is only in "essence", not in full detail.

model. NOTATION: As an alternative[4] to the notation $x \circ' y$, we often prefer to use the **bracket notation** "$[x \circ y]$".

We refer to [19, Chapter 4] for the algorithms for arithmetic in floating point systems, and about general number systems and their history. Here is a brief overview of floating point arithmetic. It involves a generic 3-step process: suppose $F = F(\beta, t)$ for simplicity, and we want to perform the binary operation $x \circ' y$ in F,

1. (Scaling) First "scale" the operands so that they share a common exponent. It makes sense to scale up the operand with the smaller magnitude to match the exponent of the operand with larger magnitude: if $x = m2^e$ and $y = n2^f$ where $e \geq f$, then scaling up means y is transformed to $(n/2^{e-f})2^e$. The scaled number may no longer be representable. Normally some truncation of bits occur.

2. (Operation) Carry out the desire operation $\circ$ on the two mantissas. This is essentially integer arithmetic.

3. (Normalization) Truncate the result of the operation back to t digits of precision. Normalize if necessary.

The Scaling Step is the key to understanding errors in floating point arithmetic: after we scale up the smaller operand, its mantissa may require much more than t digits. All hardware implementation will simultaneously truncate the scaled operand. But truncated 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 with one exception: 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 (24). This was a standard "bug" in hardware before the IEEE standard (Exercise).

Lemma 2 (Sterbenz). *Let a, b be positive normal numbers, and $\frac{1}{2} \leq \frac{a}{b} \leq 2$.*
(i) $a - b$ is representable.
(ii) If we perform subtraction of a and b using a guard digit, we get the exact result $a - b$.

Proof. Note that (ii) implies (i). To show (ii), let the normal representations of a and b be $a = a_1.a_2 \cdots a_t \times 2^{e(a)}$ and $b = b_1.b_2 \cdots b_t \times 2^{e(b)}$, where $a_1 = b_1 = 1$ and $e(a)$ denotes the exponent of the floating point representation of a. Assume $a \geq b$ (if $a < b$, then the operation concerns $-(b - a)$ where the same analysis applies with a, b interchanged). Our assumption on a/b implies that $e(a) - e(b) \in \{0, 1\}$. Consider the case $e(a) - e(b) = 1$. We execute the 3 steps of scaling, operation and normalization to compute $a - b$. To scale, we rewrite b as $0.b_1 b_2 \cdots b_t \times 2^{e(a)}$. This new representation needs $t + 1$ bits, but with the guard bit, we do no truncation. The subtraction operation has the form

$$\begin{array}{cccccc} & a_1 & . & a_2 & \cdots & a_t & 0 \\ - & 0 & . & b_1 & \cdots & b_{t-1} & b_t \\ \hline & c_0 & . & c_1 & \cdots & c_{t-1} & c_t. \end{array} \tag{25}$$

Thus $a - b = c_0.c_1 \cdots c_t \times 2^{e(a)}$. It suffices to show that $c_0 = 0$, so that after normalization, the non-zero bits in $c_1 \cdots c_t$ are preserved. Note that $a \leq 2b$; this is equivalent to $a_1.a_2 \cdots a_t \leq b_1.b_2 \cdots b_t$. Therefore $c_0.c_1 \cdots c_t = a_1.a_2 \cdots a_t - 0.b_1 b_2 \cdots b_t \leq 0.b_1 b_2 \cdots b_t$. This proves $c_0 = 0$. Note that $a - b$ might be a subnormal number. The other possibility is $e(a) = e(b)$. But this case is slightly simpler to analyze. **Q.E.D.**

Note that we cannot guarantee exact results when forming the sum $a + b$, under the assumptions of Sterbenz's Lemma. Complementing Sterbenz's lemma is another "exact" result from Dekker (1971): let $\widetilde{s}$ be the floating point result of adding a and b where we make no assumptions about a/b. Dekker shows, in base $\beta = 2$, the sum $a + b$ can be expressed exactly as $\widetilde{s} + \widetilde{e}$ where $\widetilde{e}$ is another floating point number computed from a, b. See Chapter 4.

Exercises

Exercise 2.1: Unless otherwise noted, assume the $F(2, t)$ system.
(i) Give the normal representations of $-1, 0, 1$.

[4] In our notation, "$\mathrm{fl}(x \circ y)$" is not the same as "$[x \circ y]$". The former is simply applying the rounding operator $\mathrm{fl}(\cdot)$ to the exact value $x \circ y$ while the $[\cdots \circ \cdots]$ refers to applying the floating point operation $\circ'$. The $[\cdots \circ \cdots]$ is attributed to Kahan.

(ii) Give the representations of the next representable number after 1, and the one just before 1.
(iii) Give the IEEE double formats of the numbers in (i) and (ii).
(iv) Give the binary representation of machine epsilon in the IEEE double format.
(v) True or False: for any x, $\lceil x \rceil$ and $\lfloor x \rfloor$ are the two closest representable to x. $\diamondsuit$

Exercise 2.2: 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? $\diamondsuit$

Exercise 2.3: Consider arithmetic in the system $F(\beta, t)$.
(i) Show that the standard model (24) 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 (24) 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. $\diamondsuit$

Exercise 2.4: (i) Give an actual numerical example to show why the guard digit in the scaled operands is essential for addition or subtraction.
(ii) State the error bound guaranteed by addition or subtraction when there is no guard bit. This should be weaker than the standard model. $\diamondsuit$

Exercise 2.5: (Ferguson) Generalize the lemma of Sterbenz so that the hypothesis of the lemma is that $e(a - b) \le \min\{e(a), e(b)\}$ where $e(a)$ denotes the exponent of a in normal representation. $\diamondsuit$

Exercise 2.6: 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:

No.	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 $a \ge b \ge c$. 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 parenthesis 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?

NOTE: This problem is derived from[5] Kahan's paper "Miscalculating Area and Angles of a Needle-like Triangle", July 19, 1999. ♢

End Exercises

§3. The IEEE Standard

The official name for this standard is the "IEEE Standard 754 for Binary Floating-Point Arithmetic" [16, 10]. 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 was 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 [28] for some of this history. We should properly understand the true significance of this standard vis-á-vis our robustness goal.

- It ensures consistent performance across platforms. This is no mean achievement, considering the confusion preceding its introduction and acceptance.
- Because of its rational design, it can reduce the frequency of nonrobust behavior. But we emphasize that *it does not completely eliminate nonrobustness.*
- There are issues beyond the current standard that remain to be addressed. A key problem is high-level language support for this standard [9]. 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.

In the previous section, we had presented the system $F(\beta, t, e_{\min}, e_{\max})$ that forms the mathematical foundation of floating point arithmetic. But in an actual computer implementation, we need other features that goes beyond mathematical properties. The IEEE standard provides for the following:

- Number formats. We already mentioned the single and double formats in the previous section. Another format that is expected to have growing importance is the **quadruple precision floating point format** that uses 128 bits (four computer words).
- Conversions between number formats.
- Subnormal numbers. When a computed value is smaller than the smallest normal number, we say an **underflow** has occurred. Using subnormal numbers, a computation is said to achieves a gradual underflow (see below). The older IEEE term for "subnormal numbers" is "denormal numbers".
- Special values (NaN, $\pm\infty$, ± 0). The values $\pm\infty$ are produced when an operation produces a result outside the range of our system. For instance, when we divide a finite value by 0, or when we add two very large positive values. The condition that produces infinite values is called **overflow**. But operations such as $0/0$, $\infty - \infty$, $0 \times \infty$ or $\sqrt{-1}$ result in another kind of special value called **NaN** ("not-a-number"). There are two varieties of NaN (quiet and signaling). Roughly speaking, the quiet NaN represents indeterminate operation which can propagate through the computation (without stopping). But signaling NaN represents an invalid operation (presumably this condition must be caught by the program). Similarly, the zero value has two varieties ± 0.

[5]Available from `http://cs.berkeley.edu/~wkahan/Triangle.pdf`. The CORE version of this program may be found with the CORE distribution. The above table contains a small correction of the original table in Kahan. The bug was discovered by the CORE program, and confirmed by Kahan.

- 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. The basic rule is simple enough to understand: if $\circ'$ is the machine implementation of a binary operation $\circ$, then let $x \circ' y$ should return $\mathrm{fl}(x \circ y)$, i.e., the *correctly rounded* result of $x \circ y$. This applies to unary operations as well.

- Unambiguous results for comparisons. Comparisons of finite values is not an issue. But we need to carefully define comparisons that involve special values. In particular, $+0 = -0$ and NaN is non-comparable. For instance, $NaN < x$ and $NaN \geq x$ and $NaN = x$ are all false. This means that in general, *not*$(x < y)$ and $(x \geq y)$ are unequal. Comparisons involving $\pm\infty$ is straightforward.

- 4 Rounding modes: to nearest/even (the default), up, down, to zero. These have been discussed in the previous section.

- 5 Exceptions and their handling: invalid result, overflow, divide by 0, underflow, inexact result. The philosophy behind exceptions is that many computation should be allowed to proceed even when they produce infinities, NaN's and cause under- or overflows. The special values ($\pm\infty$ and NaN) can serve to indicate such conditions in the eventual output. On the other hand, if there is a need to handle detect and handle such conditions, IEEE provide the means for this. A single arithmetic operation might cause one or more of the Exceptions to be signals. The program can trap (catch) the signals if desired. Interestingly, one of the exceptions is "inexact result", i.e., when the result calls for rounding.

Format and Encoding. Let us focus on the IEEE double precision format: how can we represent the system $F(2, 53, -1023, 1023)$ using 64 bits? The first decision is to allocate 11 bits for the exponent and 53 bits for the mantissa. To make this allocation concrete, we need to recall that modern computer memory is divided into 32-bit chunks called (computer) **words**; this is illustrated in Figure 2.

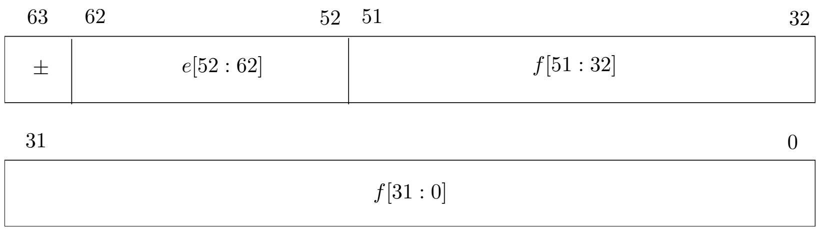

Figure 2: Format of a Double Precision IEEE Float

This physical layout will be important to understand if one were to program and manipulate such representations; So two consecutive words are used to represent a double precision number. The bits of a floating point number f are indexed by $0, 1, \ldots, 63$. The 11 exponent bits are in $f[52 : 62]$. The remaining bits, $f[0 : 51]$ and $f[63]$ are allocated to the mantissa. Bit $f[63]$ represents the sign of the mantissa. Although the sign bit logically belongs to the mantissa, it is thus physically separated from the rest of the mantissa.

We do not allocate any bits for sign in the exponent. So the 11 exponent bits represent an unsigned integer f between 0 and 2047, but we view f as encoding the signed exponent $e = f - 1023$ (here 1023 is called[6] the **exponent bias**). By avoiding a sign for the exponent, we have saved a "half-bit" using this exponent bias trick (Exercise). So e ranges from -1023 to $+1024$. We shall reserve the value $e = 1024$ for special indications (e.g., representing infinity – see below). Hence $e_{\min} = -1023$ and $e_{\max} = 1023$.

Since one of the 53 bits in the mantissa is used for sign, we have only 52 bits to represent the absolute value of the mantissa. Here, we use another trick to gain one extra bit: for normal numbers, the leading bit of a 53-bit mantissa is always 1. So this leading bit does not need to be explicitly represented! But what about subnormal numbers? In this case we declare the implicit bit to be 0. How shall we know whether a number is normal or subnormal? IEEE standard declares that this is determined by the exponent e. A number with exponent e is normal if $e > -1023$, and it is subnormal if $e = -1023$.

[6]This is distinct from the "biased exponent" idea discussed in the representation (m, e) where e is called the biased exponent.

The preceding discussion can be directly transferred to the representation of $F(2, 24, -127, 127)$ by the IEEE single precision format: with a 32-bit budget, we allocate 8 bits to the exponent and 24 bits to the mantissa. The unsigned exponent bits represents a number between 0 and 255. Using an exponent bias of 127, it encodes an exponent value e between -127 and 128. Again, the exponent $e = -127$ indicates subnormal numbers, and the value $e = 128$ is used for special indication, so $e_{\min} = -127$ and $e_{\max} = 127$.

Let us discuss how the special values are represented in the single or double precision formats. Note that ± 0 is easy enough: the mantissa shows ± 0 (this is possible since there is a dedicated bit for sign), and exponent is 0. The infinite values are represented as $(\pm 0, e_{\max})$ Finally, the NaN values are represented by $(m, e_{\max})$ where $|m| > 0$. If $|m| \geq 2^{t-1}$ then this is interpreted as a quiet NaN, otherwise it is interpreted as a signaling NaN.

■

Let us briefly mention how bits are allocated in quad-precision floating point numbers: 15 bits for the exponent and 113 bits for the mantissa. is the system $F(2, 113, -16382, 16383)$. The IEEE Standard also provide for extensions of the above types, but we shall not discuss such extensions.

Is the IEEE doubles really $F(2, 53, -1023, 1023)$? We had hinted that the IEEE double precision may not correspond exactly to $F(2, 53, -1023, 1023)$. Indeed, it is only a proper subset of $F(2, 53, -1023, 1023)$. To see this, we note that there are 2^{53} numbers in $F(2, 53, -1023, 1023)$ with exponent -1023. However, there are only 2^{52} numbers with exponent -1023 in the IEEE standard. What are the missing numbers? These are numbers of the form $\text{float}(m, -1023)$ where $|m| \geq 2^{-52}$. Why are these missing? That is because of the implicit leading bit of the 53-bit mantissa is 0 when $e = -1023$. This means m must have the form $m = 0.b_1 b_2 \cdots b_{52}$. For instance, $\text{float}(2^{-52}, -1023) = 2^{-1023}$ is not representable.

In general, all the normal numbers with exponent $e_{\min}$ are necessarily missing using the IEEE's convention. This creates an undesirable non-uniformity among the representable numbers; specifically, there is a conspicuous gap between the normal and subnormal numbers. But consider the alternative, where we get rid of the special rule concerning the implicit bit in case $e = e_{\min}$. That is, we use the rule that the implicit is always 1 even when $e = e_{\min}$. The problem is that gap between 0 and the next representable number is $2^{e_{\min}}$; the special rule for implicit leading bit gives us 2^{-t+1} values to fill this gap. So, this is where the missing numbers go! This tradeoff is apparently worthwhile, and is known as the **graceful degradation towards** 0 feature of IEEE arithmetic.

Exercises

Exercise 3.1:
(i) What are the numbers in in $F(2, 24, -127, 127)$ that are missing from the IEEE single precision floats?
(ii) In general, what are the missing numbers in the IEEE version of $F(2, t, e_{\min}, e_{\max})$?
(iii) Discuss the pros and cons of this graceful degradation towards 0 policy.
(iv) How would you would design and implementation $F(2, t, e_{\min}, e_{\max})$ where no missing numbers? ♢

Exercise 3.2: Refer to the example in Figure 1. The "missing numbers" under the IEEE scheme are those in the interval I_{-2}. The gradual underflow feature is achieved at the expense of moving these numbers into the interval $I_{-\infty}$. The result does not look pretty. Work out a scheme for more uniform floating-point grid near 0: the idea is to distribute 2^{-t+1} values uniformly in $I_{-\infty} \cup I_{\min}$ (say, filling in only the even values) What are the gap sizes in this range? ♢

Exercise 3.3: By using the biased exponent trick instead of allocating a sign bit to the exponent, what have we gained? ♢

End Exercises

§4. Error Analysis in FP Computation

The ostensible goal of error analysis is to obtain an error bound. But [14, p. 71] ("The purpose of rounding error analysis") notes that the actual bound is often less important than the insights from the analysis. The quantity denoted $\mathbf{u}$ from the previous section will an important parameter in such analysis. Without the right tool, bounding the error on the simplest computation could be formidable. The art in error analysis includes maintaining the error bounds in a form that is reasonably tight, yet easy to understand and manipulate.

Summation Problem. To illustrate this, let us analyze an extremely simple numerical code to sum the elements of an array $x[1..n]$ of numbers:

```
$s = x[1]$;
for $i = 2$ to $n$ do
    $s \leftarrow s + x[i]$
```

Let x_i be the floating point number in $x[i]$ and $s_i = \sum_{j=1}^{i} x_i$. Also let $\widetilde{s_i}$ be the value of the variable s after the ith iteration. Thus $\widetilde{s}_1 = x_1$ and for $i \geq 2$,

$$\widetilde{s_i} = [\widetilde{s}_{i-1} + x_i]$$

where we use the bracket notation "$[a \circ b]$" to indicate the floating operation corresponding to $a \circ b$. Under the standard model, we have

$$\widetilde{s_i} = (x_i + \widetilde{s}_{i-1})(1 + \delta_i) \tag{26}$$

for some δ_i such that $|\delta_i| \leq \mathbf{u}$. For instance, $\widetilde{s}_2 = (x_1 + x_2)(1 + \delta_2)$ and

$$\widetilde{s}_3 = ((x_1 + x_2)(1 + \delta_2) + x_3)(1 + \delta_3).$$

One possible goal of error analysis might be to express $\widetilde{s}_n$ as a function of s_n, n and $\mathbf{u}$. We shall see to what extent this is possible.

The following quantity will be generally useful in error analysis. Define for $n \geq 1$,

$$\gamma_n := \frac{n\mathbf{u}}{1 - n\mathbf{u}}.$$

Whenever we use γ_n, it is assumed that $n\mathbf{u} < 1$ holds. Note that $\gamma_n < \gamma_{n+1}$. The following lemma is from Higham [14, p. 69]:

Lemma 3. *Let $|\delta_i| \leq \mathbf{u}$ and $\rho_i = \pm 1$ for $i = 1, \ldots, n$. If $n\mathbf{u} < 1$ then*

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n,$$

where $|\theta_n| \leq \gamma_n$.

Proof. We use induction on n. When $n = 1$, the lemma follows from

$$(1 + \delta_1) < 1 + \frac{\mathbf{u}}{1 - \mathbf{u}}$$

and

$$(1 + \delta_1)^{-1} \leq 1 + \frac{\mathbf{u}}{1 - \mathbf{u}}.$$

Assuming the lemma for $n \geq 1$, we will prove it for $n + 1$. We consider two cases. (1) If $\rho_{n+1} = 1$, then we have

$$(1 + \theta_n)(1 + \delta_{n+1}) = 1 + \theta_{n+1}$$

or $\theta_{n+1} = \theta_n + \delta_{n+1}\theta_n + \delta_{n+1}$. Thus

$$\begin{aligned}|\theta_{n+1}| &< \gamma_n + \mathbf{u}\gamma_n + \mathbf{u}\\ &= \frac{n\mathbf{u} + n\mathbf{u}^2 + \mathbf{u}(1-n\mathbf{u})}{1-n\mathbf{u}}\\ &< \gamma_{n+1}.\end{aligned}$$

(2) If $\rho_{n+1} = -1$, we have

$$\frac{1+\theta_n}{1+\delta_{n+1}} = 1 + \theta_{n+1}$$

or $\theta_{n+1} = \frac{\theta_n - \delta_{n+1}}{1+\delta_{n+1}}$. Thus

$$\begin{aligned}|\theta_{n+1}| &\le \frac{\theta_n + \mathbf{u}}{1-\mathbf{u}}\\ &= \frac{(n+1)\mathbf{u}}{(1-n\mathbf{u})(1-\mathbf{u})}\\ &< \gamma_{n+1}.\end{aligned}$$

Q.E.D.

The hypothesis $n\mathbf{u} < 1$ is not too restrictive in typical applications. For instance, with the IEEE single precision numbers, it means $n < 2^{24}$ which is about 16 billion. If we assume $n\mathbf{u} < 1/2$, then we have the estimate $\gamma_n < 2n\mathbf{u}$. Using this lemma, it is now easy to show:

Lemma 4.
(i) $\widetilde{s}_n = \sum_{i=1}^n x_i(1+\theta_{n-i+1})$ *where* $|\theta_i| \le \gamma_i$.
(ii) If all x_i*'s have the same sign then* $\widetilde{s}_n = s_n(1+\theta)$ *where* $|\theta| \le \gamma_n$.

Proof. (i) easily follows by induction from (26). To see (ii), we note that when $xy \ge 0$ and $\alpha \le \beta$, then

$$\alpha x + \beta y = \gamma(x+y) \tag{27}$$

for some $\alpha \le \gamma \le \beta$. Thus $x_1(1+\theta_1) + x_2(1+\theta_2) = (x_1+x_2)(1+\theta)$ for some $\min\{\theta_1,\theta_2\} \le \theta \le \max\{\theta_1,\theta_2\}$. The result follows by induction. **Q.E.D.**

We noted above that the purpose of error analysis is just as important for the insight as for the actual error bound. In the present case, our proof of Lemma 4(i) shows that the backward error associated with each x_i is proportional to its depth in the expression for s_n. We can reduce the error considerably by reorganizing our computation: In particular, if the summation of the numbers $x_1, \ldots, x_n$ is organized in the form of a balanced binary tree, then we can improve Lemma 4(i) to

$$\widetilde{s}_n = \sum_{i=1}^n x_i(1+\theta_i) \tag{28}$$

where each $|\theta_i| \le \gamma_{1+\lceil \lg n \rceil}$. We leave this as an exercise.

Forward and Backward Error Analysis. Let y be a numerical variable. As a general notation, we like to write $\widetilde{y}$ for an approximation to y, and also δy for their absolute difference $\widetilde{y} - y$. Typically, $\widetilde{y}$ is some computed floating point number. There are two main kinds of error measures: absolute or relative. **Absolute error** in $\widetilde{y}$ as an approximation to y is simply $|\delta y|$. **Relative error** is defined by

$$\varepsilon(\widetilde{y}, y) = \frac{|\delta y|}{|y|}.$$

This is defined to be 0 if $y = \widetilde{y} = 0$ and ∞ if $y = 0 \ne \widetilde{y}$. Generally speaking, numerical analysis prefers to use relative error measures. One reason is that relative error for floating point numbers is built-in; this is clear from Theorem 1.

In error analysis, we also recognized two kinds of algorithmic errors: forward and backward errors. Let $f : X \to Y$ be a function with $X, Y \subseteq \mathbb{C}$. Suppose $\widetilde{y}$ is the computed value of f at $x \in Y$ and $y = f(x)$. How shall we measure the error in this computation? Conceptually, forward error is simple to understand – it measures how far off our computed value is from the true value. Again, this can be absolute or relative: so the **absolute forward error** is $|\delta y|$ and the **relative forward error** is $\varepsilon(\widetilde{y}, y)$. The backward error is how far is x from the input $\widetilde{x}$ for which $\widetilde{y}$ is the exact solution. More precisely, the **absolute backward error** of $\widetilde{y}$ is defined to be the infimum of $|\delta x| = |\widetilde{x} - x|$, over all $\widetilde{x}$ such that $\widetilde{y} = f(\widetilde{x})$. If there is no such $\widetilde{x}$, then the absolute backward error is defined to be ∞. The **relative backward error** of $\widetilde{y}$ is similarly defined, except we use $\varepsilon(\widetilde{x}, x)$ instead of $|\delta x|$.

In general, X, Y are normed spaces. The forward error is based on the norm in range space Y, while backward error is based on the norm in domain space X. Consider our analysis of the summation problem, Lemma 4. The computed function is $f : \mathbb{C}^n \to \mathbb{C}$. Also, let take the ∞-norm in $X = \mathbb{C}^n$. Part (i) gives us a relative backward error result: it says that the computed sum $\widetilde{s}_n$ is the correct value of inputs that are perturbed by at most γ_n. Part (ii) gives us a forward error result: it says that the relative error $\varepsilon(\widetilde{s}_n, s_n)$ is at most γ_n.

It turns out that backward error is more generally applicable than forward error for standard problems of numerical analysis. Let us see why. Note that Lemma 4(ii) has an extra hypothesis that the x_i's have the same sign. How essential is this? In fact, the hypothesis cannot be removed even for $n = 2$. Suppose we want to compute the sum $x + y$ where $x > 0$ and $y < 0$. Then we do not have the analogue of (27) in case $x + y = 0$. Basically, the possibility of cancellation implies that no finite relative error bound is possible.

Less the reader is lulled into thinking that backward error analysis is universally applicable, we consider the example [14, p. 71] of computing the outer product of two vectors: $A = xy^T$ where x, y are n-vectors and $A = (a_{ij})_{i,j} = (x_i y_j)_{i,j}$ is a $n \times n$-matrix. Let $\widetilde{A} = (\widetilde{a}_{ij})$ be the product computed by the obvious trivial algorithm. The forward analysis is easy because $\widetilde{a}_{ij} = a_{ij}(1 + \theta)$ where $|\theta| \le \mathbf{u}$. But there is no backwards error result because $\widetilde{A}$ cannot be written as $\widetilde{x}\widetilde{y}^T$ for any choice of $\widetilde{x}, \widetilde{y}$, since we cannot guarantee that $\widetilde{A}$ is a rank-one matrix.

In general, we see that for problems $f : X \to Y$ in which the range $f(X)$ is a lower dimensional subset of Y, no backward error analysis is possible. Standard problems of numerical analysis are usually not of this type.

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 return to this issue.

Exercises

Exercise 4.1: The inequality $|\theta_n| \le \gamma_n$ in Lemma 3 is actually strict, with one possible exception. What is this? ♢

Exercise 4.2: Extend the error analysis for summation to the scalar product problem: $s = \sum_{i=1}^n x_i y_i$. ♢

Exercise 4.3: Write a recursive program for summing the entries in the array $x[1..n]$ so that the error bound (28) is attained. Also, prove the bound (28) for your scheme. ♢

Exercise 4.4: We have noted that the relative forward error analysis for $s = \sum_{i=1}^n x_i$ is not possible in general. Do the absolute forward error analysis. ♢

Exercise 4.5: Suppose we introduce a "composite" error combining relative with absolute. More precisely, let $\varepsilon(\widetilde{y}, y) = \min\{|y - \widetilde{y}|, |y - \widetilde{y}|/|y|\}$. Thus, in the neighborhood around $y = 0$, it is the absolute error that takes effect, and in the neighborhood of $|y| = \infty$, the relative error takes effect. Show that we can how provide a forward error analysis for the general summation problem. ♢

End Exercises

§5. Condition Number and Stability

Despite its acknowledged importance in numerical analysis, the concept of "stability" has largely remained an informal notion. The book of Higham [14], for instance, says [14, p. 8] that an algorithm for $y = f(x)$ is "backward stable" if the backward error $\varepsilon(\widetilde{x}, x)$ is "small" in some context-specific sense. The book of Trefethen and Bau [33] devoted several chapters to stability, and offers the most explicit definition of stability. This is reproduced below. Before delving into stability, let us consider the more primitive concept: the condition number.

Suppose

$$f : X \to Y \tag{29}$$

where X, Y are normed vector spaces. If we want to emphasize the distinction between these two norms, we may write $\|\cdot\|_X$ and $\|\cdot\|_Y$. But generally, we will omit the subscripts in the norms. For $x \in X$, we again use our convention of writing $\widetilde{x}$ for some approximation to x and

$$\delta x := \widetilde{x} - x.$$

The condition number of f at $x \in X$ is a non-negative number (possibly infinite) that measures the sensitivity of $f(x)$ to x; a larger condition number means greater sensitivity. Again we have the absolute and relative versions of condition number.

The **absolute condition number** is defined as

$$\widehat{\kappa}_f(x) := \lim_{\delta \to 0} \sup_{\|\delta x\| \le \delta} \frac{\|f(\widetilde{x}) - f(x)\|}{\|\delta x\|}.$$

The **relative condition number** is defined as

$$\begin{aligned} \kappa_f(x) \quad &:= \quad \lim_{\delta \to 0} \sup_{\|\delta x\| \le \delta} \left(\frac{\|f(\widetilde{x}) - f(x)\|}{\|f(x)\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right) \\ &:= \quad \lim_{\delta \to 0} \sup_{\|\delta x\| \le \delta} \left(\frac{\|f(\widetilde{x}) - f(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|f(x)\|} \right). \end{aligned} \tag{30}$$

Viewing $\widehat{\kappa}, \kappa : X \to \mathbb{R}_{\ge 0} \cup \{\infty\}$ as functions, we call these the **condition number functions** of f. When f is understood, we may drop the subscripts from the κ-notations. REMARK: we could also consider how an absolute error in X affects the relative error in Y, or how a relative error in X affects the absolute error in Y. This gives rise to 2 additional concepts of relative error.

Suppose $X \subseteq \mathbb{C}^m$, $Y \subseteq \mathbb{C}^n$ and $f = (f_1, \ldots, f_n)$ is differentiable. Let $J_f(x)$ be the **Jacobian** of f at x: this is a $n \times m$ matrix whose (i,j)-th entry given by $\partial f_i / \partial x_j$. The norm $\|J_f(x)\|$ on the Jacobian will be that induced by the norms in X and Y, namely,

$$\|J_f(x)\| = \sup_x \|J_f(x) \cdot x\|_Y$$

where x range over those elements of X satisfying $\|x\|_X = 1$. Then the absolute and relative condition numbers are given by

$$\widehat{\kappa}_f(x) = \|J_f(x)\| \tag{31}$$

and

$$\kappa_f(x) = \frac{\|J_f(x)\| \cdot \|x\|}{\|f(x)\|}. \tag{32}$$

EXAMPLE 1. Let us give a proof of (32) for the case $f : \mathbb{C}^m \to \mathbb{C}$. By Taylor's theorem with remainder,

$$f(x + \delta x) - f(x) = J_f(x) \circ \delta x + R(x, \delta x)$$

where $J_f(x) \circ \delta x$ denotes dot product with $J_f(x) = [\partial f / \partial x_1, \ldots, \partial f / \partial x_n]$, and $\frac{|R(x,\delta x)|}{\|\delta x\|} \to 0$ as $\|\delta x\| \to 0$. Taking absolute values and multiplying by $\frac{\|x\|}{|f(x)| \cdot \|\delta x\|}$, we get

$$\frac{|f(x + \delta x) - f(x)|}{\|\delta x\|} \cdot \frac{\|x\|}{|f(x)|} \quad = \quad \frac{\|x\|}{|f(x)|} \left(\left| J_f(x) \circ \frac{\delta x}{\|\delta x\|} \right| + \frac{|R(x, \delta x)|}{\|\delta x\|} \right).$$

Taking the limsup of the lefthand side as $\|\delta x\| \to 0$, we obtain $\kappa_f(x)$. But the righthand side equals $\frac{\|x\|}{|f(x)|}\|J_f(x)\|$, by definition of $\|J_f(x)\|$. This completes our demonstration.

EXAMPLE 2. Let us compute these condition numbers for the problem $f : \mathbb{C}^2 \to \mathbb{C}$ where $f(x) = x_1 + x_2$ and $x = (x_1, x_2)^T$. The Jacobian is

$$J_f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right] = [1, 1]$$

Assume the norm on $X = \mathbb{C}^2$ is the ∞-norm, so $\|x\| = \max\{|x_1|, |x_2|\}$. Thus $\|J_f(x)\| = 2$. From (31), we obtain $\widehat{\kappa}_f(x) = 2$. From (32), we obtain

$$\kappa = \frac{2\max\{|x_1|, |x_2|\}}{|x_1 + x_2|}.$$

Thus $\kappa = \infty$ when $x_1 + x_2 = 0$.

EXAMPLE 3. Condition numbers of matrix.

EXAMPLE 4. Condition numbers of polynomial root.

Stability of Algorithms. Condition numbers are inherent to a given problem, the concept of stability is a function of algorithms for the problem. We may view an algorithm for the problem (29) as another function $\widetilde{f} : X' \to Y'$ where $X' \subseteq X$ and $Y' \subseteq Y$. In fact, we might as well take $X' = \mathrm{fl}(X)$ and $Y' = \mathrm{fl}(Y)$, i.e., the representable elements of X, Y. The stability of $\widetilde{f}$ is the measure of how much $\widetilde{f}$ deviates from f. Intuitively, we want to define $\widetilde{\kappa}_f(x)$, the relative condition number of $\widetilde{f}$ at x; we need to be careful with this definition since $X' = \mathrm{fl}(X), Y' = \mathrm{fl}(Y)$ are discrete sets. The stability of an algorithm is clearly limited by the condition numbers for the problem. Ideally, a stable algorithm should have the property that $\widetilde{\kappa}_f(x) = O(\kappa_f(x))$.

We now come to the key definition of Trefethen and Bau: an algorithm $\widetilde{f}$ for f is **stable** if there exists positive constants $C_0, C_1, \mathbf{u}_0$ such that for all $\mathbf{u}$ $(0 < \mathbf{u} < \mathbf{u}_0)$ and all $x \in X$, there is some $\widetilde{x} \in X'$ such that

$$\frac{\|\widetilde{x} - x\|}{\|x\|} \le C_0\mathbf{u}, \quad \text{and} \tag{33}$$

$$\frac{\|\widetilde{f}(x) - f(\widetilde{x})\|}{\|f(\widetilde{x})\|} \le C_1\mathbf{u}. \tag{34}$$

According to Trefethen and Bau, the constants C_0, C_1 may not depend on x but may depend on f. In practice, the requirement that $\mathbf{u}$ must be smaller than some $\mathbf{u}_0$ is not an issue. To understand this definition of stability, let us explore some related concepts:

- The requirement (34) is a little subtle: for instance, it might be more obvious to require

$$\frac{\|\widetilde{f}(x) - f(x)\|}{\|f(x)\|} \le C_1\mathbf{u}. \tag{35}$$

  This would not involve $\widetilde{x}$, so the requirement (33) can be omitted. Of course, (35) is the relative forward error in $\widetilde{f}(x)$; in fact, this constitutes our definition of **forward stability**.

- If, in the definition of stability above, the constant C_1 is chosen to be 0, we say the algorithm $\widetilde{f}$ is **backward stable** for f. In other words, we can find $\widetilde{x}$ subject to (33) *and* $\widetilde{f}(x) = f(x)$.

- In contrast to forward or backward stability, the view of (34) is to compare $\widetilde{f}(x)$ to the "correct solution $f(\widetilde{x})$ of a nearly correct question $\widetilde{x}$". Thus our concept of stability contains a mixture of forward and backward error concepts.

- We may rewrite equations (33) and (34) using the standard big-Oh notations:

$$\frac{\|\widetilde{x} - x\|}{\|x\|} = O(\mathbf{u}),$$

$$\frac{\|\widetilde{f}(x) - f(\widetilde{x})\|}{\|f(\widetilde{x})\|} = O(\mathbf{u}).$$

In the big-Oh notations of equations (33) and (34), we view $\mathbf{u}$ as varying and approaching 0. As noted above, the implicit constants C_0, C_1 in these big-Oh notations do not depend on x but may depend on f. Typically, $X \subseteq \mathbb{C}^m$ for some fixed m. Thus the implicit constants C_0, C_1 are allowed to depend on m. When $X = \cup_{m \geq 1} \mathbb{C}^m$, then it seems that we should allow the constant C_0, C_1 to have a weak dependence of x: in particular, we would like C_0, C_1 to depend on size(x) where size(x)=m when $x \in \mathbb{C}^m$.

EXAMPLE 5: From theorem 1, we conclude that the standard algorithms for performing arithmetic operations $(+, -, \times, \div)$ are all stable.

EXAMPLE 6: Consider the problem of computing eigenvalues of a matrix A. One algorithm is to compute the characteristic polynomial $P(\lambda) = \det(A - \lambda I)$, and then find roots of $P(\lambda)$. Trefethan and Bau noted that this algorithm is not stable.

There is a rule of thumb in numerical analysis that says

$$\text{Forward Error} \leq \text{Condition Number} \times \text{Backward Error}.$$

Here is one precise formulation of this insight.

Theorem 5 ([33, p. 151]). *If $\widetilde{f}$ is a backward stable algorithm for f then the relative forward error satisfy*

$$\frac{\|\widetilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\kappa_f(x)\mathbf{u}).$$

Proof. By definition of $\kappa = \kappa_f(x)$, for all $\varepsilon > 0$ there exists $\delta > 0$ such that

$$\sup_{\|\delta x\| \leq \delta} \left(\frac{\|f(x+\delta) - f(x)\|}{\|\delta x\|} \cdot \frac{\|x\|}{\|f(x)\|} \right) \leq \kappa + \varepsilon. \tag{36}$$

By definition of backward stability, for all $\mathbf{u} < \mathbf{u}_0$ and all $x \in X$, there is δx such that

$$\frac{\|\delta x\|}{\|x\|} = O(\mathbf{u}) \tag{37}$$

and

$$\widetilde{f}(x) = f(x + \delta x). \tag{38}$$

Plugging (38) into (36), we conclude

$$\begin{array}{lll} \frac{\|\widetilde{f}(x)-f(x)\|}{\|\delta x\|}\frac{\|x\|}{\|f(x)\|} & \leq & \kappa + \varepsilon \\ \frac{\|\widetilde{f}(x)-f(x)\|}{\|f(x)\|} & \leq & O(\kappa)\frac{\|\delta x\|}{\|x\|} \quad (\text{choose } \varepsilon = O(\kappa)) \\ & = & O(\kappa\mathbf{u}) \quad (\text{by } (37)). \end{array}$$

Q.E.D.

Corollary 6. *If the condition number function of $f : X \to Y$ is bounded, and $\widetilde{f}$ is a backward stable algorithm for f, then $\widetilde{f}$ is a forward stable algorithm for f.*

Exercises

Exercise 5.1: Give the proof (32) in the general case of $f : \mathbb{C}^m \to \mathbb{C}^n$. ♢

Exercise 5.2: (Trefethan and Bau) Compute $\kappa_f(x)$ for the following functions.
(i) $f : \mathbb{C} \to \mathbb{C}$ where $f(x) = x/2$.
(ii) $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = \sqrt{x}$ and $x > 0$.
(iii) $f : \mathbb{R} \to \mathbb{R}$ where $f(x) = \tan(x)$ and $x = 10^{100}$. ♢

Exercise 5.3: Show that the algorithm for eigenvalues in EXAMPLE 6 is not stable. HINT: it is enough to show this for the case where A is a 2×2 matrix: show that the error is $\Omega(\sqrt{\mathbf{u}})$. ♢

End Exercises

§6. Arbitrary Precision Computation

In contrast to fixed precision arithmetic, we now consider number types with no *á priori* bound on its precision, save for physical limits on computer memory, etc. We call[7] this **arbitrary precision computation** (or "AP computation") Arbitrary precision computation is becoming more mainstream, but it is still a small fraction of scientific computation. Unlike the ubiquitous hardware support for FP computation, AP computation is only available through software. It is the computational basis for the fields of computer algebra, computational number theory and cryptographic computations. 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 for manipulating and performing arithmetic with such numbers are usually called **Big Number Packages**. 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**. 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 standard. 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 $0 \le a_i < \beta$ for some natural number $\beta > 1$). The third 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

$$m\beta^e \tag{39}$$

(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 (39) is at variance with the system $F(\beta, t)$ introduced in Lecture 2. If we want an analogous system, we can can proceed as follows: assuming that m is also represented in base β, let us define

$$\mu(m) \leftarrow \lceil \log_\beta(|m|) \rceil .$$

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

$$Float(m, e) := \beta^{e-\mu(m)}. \tag{40}$$

Call this the **normalized Big Float** value. Since $\beta^{-1} < m\beta^{-\mu(m)} \le 1$, 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; if these are equal, we then compare their mantissas.

Arbitrary precision arithmetic can also be based on Hensel's p-adic numbers [12, 13, 8], or on continued fractions.

On Rationals versus Integers versus Big Floats. Conceptually, Big Rationals does not appear to be different from Big Integers. It is less well-known that from a complexity viewpoint, Big Rational computations are 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 [18] reported that a straightforward rational implementation of determinants can cause a slow down of 5 orders of magnitude. In their paper, they describe filtering techniques which finally bring the slowdown down to a small constant. This is one of the first evidence of the importance of filters in geometric problems.

[7] Alternatively, "multi-precision computation" or "any precision". However, we avoid the term "infinite precision" since it engenders some confusion.

On the other hand, Big Float arithmetic is considerably faster than Big Rational arithmetic. *Big Floats recover most 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.

§7. Interval Arithmetic

This leads us to the concept of **significance arithmetic**. First let us clarify the notion of "significance". Assuming the normalized Big Float system (40), 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, then triple (m, e, u) represents the interval

$$[\text{float}(m-u, e), \text{float}(m+u, e)].$$

In this case, the significance of (m, e, u) is defined to be $\ell(m) - \ell(u)$. This means that we can only trust the first $\ell(m) - \ell(u)$ digits of float(m, e). Significance arithmetic is usually traced to the work of Metropolis [2, 23]. Significant arithmetic can be regarded as a special case of interval arithmetic, where we attempt to only retain the uncertainty warranted by the input uncertainty. Basically, interval arithmetic [25] replacing each number x by an interval $I = [a, b]$ that contains x. We can introduce the basic arithmetic operations on such intervals in the natural way (see below). The development and analysis of techniques for computation with intervals constitute the field of **interval arithmetic**. Taking a larger view, the field is also[8] known as **validated computation**. R.E. Moore [25] is the pioneer of interval arithmetic. The book of Rokne and Ratschek [29] gives an excellent account of interval functions.

A motivation for interval analysis is the desire to compute error bounds efficiently and automatically. In [1], this is called the "naïve outlook"; instead, it is suggested that the proper focus of the field ought to be how to compute with inexact data (e.g., solving a linear system with interval coefficients) and for problems with exact data, the issues of algorithmic convergence.

¶1. For $D \subseteq \mathbb{R}$, let $I(D)$ denote the set of closed intervals of the form $[a, b] \in D$ where $a \leq b$. An alternative notation for $I(D)$ is $\square D$. The n-fold Cartesian product of $I(D)$ is $I(D)^n$ or $(\square D)^n$. For simplicity, we also write $I^n(D)$ and $\square^n D$. Elements of $I^n(D)$ are called **boxes**; when $n = 1$, we prefer to say "intervals" instead of boxes.

Let $\ell(I)$ and $u(I)$ denote the lower and upper endpoints of interval I. So $I = [\ell(I), u(I)]$. If $\ell(I) < u(I)$, we call I a **proper interval**; otherwise I is **improper**. Thus $\mathbb{R}$ is regarded as contained in $I(\mathbb{R})$ by identifying elements of $\mathbb{R}$ with the improper intervals.

We now extend the basic operations and predicates on real numbers to intervals. But it is useful to see them as special cases of definitions which apply to arbitrary sets of real numbers. In the following, let $A, B \subseteq \mathbb{R}$.

1. We can compare A and B in the natural way: we write $A \leq B$ to mean that the inequality $a \geq b$ holds for all $a \in A, b \in B$. The relations $A \leq B$, $A > B$ and $A < B$ are similarly defined. If $A \leq B$ or $A \geq B$, we say A and B are **comparable**; otherwise they are **incomparable**. In case A, B are intervals, these relations can be reduced to relations on their endpoints. For instance, $A \leq B$ iff $u(A) \leq \ell(B)$.

2. If $\circ$ is any of the four arithmetic operations then $A \circ B$ is defined to be $\{a \circ b : a \in A, b \in B\}$. It is assume that in case of division, $0 \notin B$. In case A, B are intervals, it is easy to see that $A \circ B$ would be intervals. Moreover, $A \circ B$ can be expressed in terms of operations on the endpoints of A and B:

   - $A + B = [\ell(A) + \ell(B), u(A) + u(B)]$
   - $A - B = [\ell(A) - u(B), u(A) - u(B)]$
   - $A \cdot B = [\min S, \max S]$ where $S = \{\ell(A)\ell(B), \ell(A)u(B), u(A)\ell(B), u(A)u(B)\}$
   - $1/B = [1/u(B), 1/\ell(B)]$

[8]Instead of "validated", such computations are also described as "certified" or "verified", and sometimes "guaranteed". The error bounds in such computations are á posteriori ones. We prefer to reserve the term "guaranteed" for the stronger notion of á priori error bounds.

- $A/B = A \cdot (1/B)$

3. If $f : \mathbb{R}^n \to \mathbb{R}$ is a real function and $A \subseteq \mathbb{R}^n$, we define $f(A) = \{f(a) : a \in A\}$. In general, $f(A)$ may not be an interval even if A is a box. We call a function $\square f$ of the form

$$\square f : \square^n \mathbb{R} \to \square \mathbb{R}$$

a **box function** for f provided it satisfies two basic properties:
(i) [Inclusion property] For all boxes $B \in \square^n \mathbb{R}$, $f(B) \subseteq \square f(B)$.
(ii) [Convergence property] If $B_i \subseteq \square^n \mathbb{R}$ $(i \in \mathbb{N})$ is a sequence of boxes that converges to a point $p \in \mathbb{R}^n$, then $\square f(B_i)$ converges to $f(p)$.

One way to obtain a box function is to define

$$\square f(B) = MBB(f(B))$$

where $MBB(S)$ denotes for the minimum bounding box for set $S \subseteq \mathbb{R}^n$. This box function is clearly "optimal" but quite impractical. Below we shall discuss practical construction of box functions.

Another commonly satisfied property of box functions is (inclusion) **isotony**: $B \subseteq B'$ implies $\square f(B) \subseteq \square f(B)$. It is easy to check that the four arithmetic operations implemented in the standard way is inclusion isotone. It follows that if $\square f$ is defined by the interval evaluation of any rational expression defining f.

¶2. Algebraic Properties. The set $I(\mathbb{R})$ under these operations has some of the usual algebraic properties of real numbers: $+$ and $\times$ are both commutative and associative, and have $[0, 0]$ and $[1, 1]$ (respectively) as their unique identity elements, and there are no zero-divisors. We have distributivity $A(B+C) = AB+BC$ provided $B \times C \geq 0$. In general, we have **subdistributivity**, as expressed by the following lemma:

Lemma 7 (Subdistributivity Property). *If A, B, C are intervals, then $A(B + C) \subseteq AB + AC$.*

However, no proper intervals has any inverse under $+$ or $\times$. A critical property is **monotonicity**: if $A \subseteq A'$ and $B \subseteq B'$ then $A \circ B \subseteq A' \circ B'$. This is also called isotonicity.

¶3. Mid-point Representation. 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. It is useful to have notations, $m(I) = (a + b)/2$ and $w(I) = b - a$, to denote the **midpoint** and **width** of $I = [a, b]$. But we can also have the **mid-point representation** in which a pair (x, u) of numbers represents the interval $[x - u, x + u]$. Think of u as the **uncertainty** (or **radius**) about the **representative value** of x. Clearly, the interval $I = [a, b]$ is represented by $(m(I), w(I)/2)$ in this representation.

If $B \in \square^n \mathbb{R}$ where $B = I_1 \times \cdots \times I_n$, we can also define $m(B) = (m(I_1), \ldots, m(I_n))$, and $w(B) = \max_{i=1}^n w(I_i)$. In case $I_1 = I_2 = \cdots = I_n$, we can regard the pair $(m(B), w(I_1)/2)$ as a midpoint representation of B.

What can we gain by using the mid-point representation? If the desired interval is small, then u may need only a few bits to represent. Then the pair (x, u) may save up to half of the number of bits in the end-point representation. Rump [30] has pointed out that the midpoint representation is uniformly bounded by a factor of 1.5 in optimum radius for the 4 basic arithmetic operations as well as for vector and matrix operations over reals and complex numbers. Moreover, this can take advantage of vector and parallel architectures. We leave it as an exercise to work out the implementation of arithmetic based on midpoint representations. In the original Core Library, our BigFloat numbers represent intervals in the midpoint representation.

¶4. Complex Intervals. Most of our discussion concern real intervals. There is the obvious and simple of interval arithmetic to complex numbers: a pair $[a, b]$ of complex numbers represents the complex "interval" $\{z \in \mathbb{C} : a \leq z \leq b\}$ where $a \leq b$ means that $\mathtt{Re}(a) \leq \mathtt{Re}(b)$ and $\mathtt{Im}(a) \leq Im(b)$. Many of our discussions generalize directly to this setting. One generalization of the midpoint representation to complex numbers introduces the geometry of balls: given $z \in \mathbb{C}, r \geq 0$, the pair (z, r) can be viewed as representing the ball $\{w \in \mathbb{Z} : |w - z| \leq r\}$. We shall not say much more about complex intervals.

¶5. Box Functions. We consider the problem of constructing box functions. It is easy to see that the basic arithmetic operations ($\pm, \times, \div$) defined above are box functions. It follows that if f is any rational function, and we define $\square f(B)$ by evaluating a fixed rational expression for f, then the result is a box function for f. Suppose f is a univariate polynomial, $f = \sum_{i=0}^{m} a_i X^i$. Here are some possibilities.

- In most applications, we may restrict ourselves to intervals $I \in \square\mathbb{R}$ with a definite sign: i.e., either $I = 0$ or $I > 0$ or $I < 0$. Let us assume $I > 0$ in the following; the case $I < 0$ is treated analogously, and the case $I = 0$ is trivial. We write $f = f^+ - f^-$ where f^+ is just the sum of those terms of f with positive coefficients. Then we may define $f^- := f^- - f$. If $I = [a, b]$, then define the box function
$$\square_1 f(I) = [f^+(a) - f^-(b), f^+(b) - f^- a].$$
It is not hard to verify that $\square_1 f$ is a box function.

- We can also define box functions by specifying a fully parenthesized expression E for f. For instance, Horner's rule for evaluating f gives rise to the expression
$$E = (\cdots((a_m X + a_{m-1})X + a_{m-2}X + \cdots + a_0).$$
Now, we can define the box function $\square_E f(I)$ which returns the interval if we evaluate f on I using the expression E. For instance, if $f = 2X^2 - 3X + 4$ then Horner's expression for f is $E = ((2X - 3)X + 4)$. If $I = [1, 2]$ then
$$\begin{aligned} \square_E f(I) &= ((2I - 3)I + 4) = (([2, 4] - 3)[1, 2] + 4) \\ &= ([-1, 1][1, 2] + 4) = [-2, 2] + 4] = [2, 6]. \end{aligned}$$
We will denote the box function by $\square_0 f$ where E is Horner's expression for f.

- Consider a third way to define box functions, where E is basically the expression given by the standard power basis of f: namely, we evaluated each term of f, and sum the terms. Call the corresponding box function $\square_2 f$. We can show that $\square_2 f$ is the same as $\square_1 f$. Moreover, for all $I \in \square\mathbb{R}$,
$$\square_0 f(I) \subseteq \square_1 f(I).$$
This follows from the subdistributivity property of interval arithmetic.

In some applications, box functions suffice. E.g., Plantinga and Vegter (2004) shows that the isotopic approximation of implicit non-singular surfaces can be achieved using box functions. Sometimes, we want additional properties. For instance, the ability to specify a precision parameter $n > 0$ such that
$$\square f(B; n) \subseteq MBB(f(B) \oplus E_n)$$
where $\oplus$ is Minkowsky sum, $E_n = (I_n)^d$ where $I_n = [-2^{-n}, 2^n]$.

EXAMPLE 1. Suppose we want to solve the equation $AX = B$ where $A = [a, a'] \neq 0$. Define $\chi(A) = a/a'$ if $|a| \leq |a'|$, and $\chi(A) = a'/a$ otherwise. There is a solution interval $X \in I(\mathbb{R})$ iff $\chi(A) \geq \chi(B)$. Moreover the solution is unique unless $\chi(A) = \chi(B) \leq 0$.

We introduce the Hausdorff metric on $I(\mathbb{R})$ by defining $d(A, B) = \max\{|a - b|, |a' - b'|\}$ where $A = [a, a'], B = [b, b']$. This is a metric because $d(A, B) \geq 0$ with equality iff $A = B$, $d(A, B) = d(B, A)$ and finally $d(A, C) \leq d(A, B) + d(B, C)$. A sequence of intervals **converges** to some interval $A = [a, a']$ iff the left and right endpoints of the interval sequence converges to a and a' respectively. Under this metric, we can define the concept of continuity and show that the four arithmetic operations are continuous.

Let $f(x)$ be a real function. If $X \in I(\mathbb{R})$, we define $f(x) = [\underline{a}, \overline{a}]$ where $\underline{a} = \min_{x \in X} f(x)$ and $\overline{a} = \max_{x \in X} f(x)$. Let $E(x), E'(x)$ be two real expressions which evaluates to $f(x)$ when the input intervals are improper. The fact that certain laws like commutativity fails for intervals means that E and E' will in general obtain different results when we evaluate them at proper intervals.

EXAMPLE 2. Let the function $f(x) = x - x^2$ be computed by the two expressions $E(x) = x - x^2$ and $E'(x) = x(1-x)$. When x is replaced by the interval $X = [0, 1]$ then $f([0, 1]) = \{x - x^2 : 0 \leq x \leq 1\} = [0, 1/4]$. But $E([0, 1]) = [0, 1] - [0, 1]^2 = [0, 1] - [0, 1] = [-1, 1]$ and $E'([0, 1]) = [0, 1](1 - [0, 1]) = [0, 1][0, 1] = [0, 1]$.

In fact, we have the following general inclusion:

$$f(X) \subseteq E(X)$$

for all $X \in I(\mathbb{R})$. In proof, note that every $y \in f(X)$ has the form $y = f(x)$ for some $x \in X$. But it is clear that the value $f(x)$ belongs to $E(X)$, since it can be obtained by evaluating $E(X)$ when every occurrence of X in $E(X)$ is replaced by x.

A simpler example is $E(x) = x - x$ and $f(x) = x - x$. If $X = [0, 1]$ then $E(X) = [-1, 1]$ while $f(X) = 0$ (in fact, $f(Y) = 0$ for any interval Y).

¶6. Rounding and interval arithmetic. Machine floating point numbers can be used in interval arithmetic provided we can control the rounding mode. In the following, assume machine numbers are members of $F(\beta, t)$ for some base $\beta > 1$ and precision $t \geq 1$. We need 2 kinds of rounding: for any real number x, let round up $\mathrm{fl}_{up}(x)$ and round down $\mathrm{fl}_{down}(x)$ be the closest numbers in $F(\beta, t)$ such that $\mathrm{fl}_{down}(x) \leq x \leq \mathrm{fl}_{up}(x)$. Then $A = [a, b] \in I(\mathbb{R})$ can be **rounded** as $\mathrm{fl}(A) = [\mathrm{fl}_{down}(a), \mathrm{fl}_{up}(b)]$. If we view $x \in \mathbb{R}$ as an interval, we now have $fl(x) = [\mathrm{fl}_{down}(x), \mathrm{fl}_{up}(x)]$. If $x \in F(\beta, t)$ and the exponent of x is e then the width of $fl(x)$ is β^{-t+e}. Rounding can be extended to the arithmetic operations in the obvious way: if $\circ \in \{+, -, \times, \div\}$ and $\circ'$ is the interval analogue, we define $A \circ' B := fl(A \circ B)$. Inclusion monotonicity is preserved: If $A \subseteq A', B \subseteq B'$ then $A \circ' B \subseteq A' \circ' B'$.

In practice, it is inconvenient to use two rounding modes within a computation. One trick is to store the interval $A = [a, b]$ as the pair $(-a, b)$ and use only round up. Then $\mathrm{fl}(A)$ is represented by the pair $(\mathrm{fl}_{up}(-a), \mathrm{fl}_{up}(b))$.

¶7. Machine arithmetic. One class of results in interval analysis addresses the question: suppose we compute with numbers in $F(\beta, t)$. This is a form of idealized machine arithmetic in which we ignore issues of overflow or underflow. How much more accuracy do we gain if we now use numbers in $F(\beta, t')$ where $t' > t$? If $A = [a, b]$, let the **width** of A be $w(A) := b - a$. $w(\mathrm{fl}(x)) \leq \beta^{e-t}$ where e is the exponent. Any real number x can be represented as

$$x = \beta^e \left(\sum_{i=1}^{\infty} d_i \beta^{-i} \right)$$

with $0 \leq d_i \leq \beta - 1$. This representation is unique provided it is not the case that each digit $d_i = \beta - 1$ for all i beyond some point. Then $\mathrm{fl}_{down}\, x = \beta^e \left(\sum_{i=1}^{t} d_i \beta^{-i} \right)$ and $\mathrm{fl}_{up}\, x \leq \mathrm{fl}_{down}(x) + \beta^{e-t}$. Hence $w(\mathrm{fl}(x)) \leq \beta^{e-t}$. The following is a basic result (see [1, theorem 5, p.45]).

Theorem 8. *Let an algorithm be executed using machine arithmetic in $F(\beta, t)$ and also in $F(\beta, t')$ for some $t' > t$. Assuming both computations are well-defined, then the relative and absolute error bounds for the result is reduced by a factor of $\beta^{t-t'}$ in the latter case.*

¶8. Lipschitz Condition. An interval function $\square f : \square^n D \to \mathbb{R}$ is **Lipschitz** with **Lipschitz constant** $K > 0$ if

$$w(\square f(I)) \leq K w(I), \qquad I \in \square^n D.$$

A box function $\square f$ is **continuous** if B_i $(i \geq 0)$ is a sequence of boxes converging to some B; here convergence is in the sense that the left (right) endpoints of the B_i's converges to the left (right) endpoint of B. For real functions, being Lipschitz is equivalent to being continuous. But this breaks down for box functions:

Lipschitz may not be continuous: suppose $\delta(x) = 1$ if $x > 0$ and $\delta(x) = 0$ otherwise. Let $\square f : \square\mathbb{R} \to \square\mathbb{R}$ be given by $\square f(B) = B + \delta(w(B))$. Then $w(\square(B)) = w(B)$. and $\square f$ is Lipschitz. However, $\square f$ is not continuous at 0 since the sequence $B_n = [0, 1/n]$ converges to 0 but the sequence $\square f(B_n)$ converges to 1.

Continuous functions may not be Lipschitz: With $B = [0, 1]$, let $\square f : I(B) \in B$ be the function $\square f(I) = [0, \sqrt{w(I)}]$. Then $\square f$ is continuous but not Lipschitz in $I(B)$ because the sequence of quotients

$$w(\square f(I))/w(I) = \sqrt{w(I)}/w(I) = 1/\sqrt{w(I)}, \qquad w(I) \neq 0$$

is unbounded.

¶9. Quadratic convergence and centered forms. Given a function $f : \mathbb{R}^n \to \mathbb{R}$, we want to compute a box function $\square f : \square^n\mathbb{R} \to \mathbb{R}$ such that $w(\square f(B))$ is "small relative to $w(B)$". One interpretation is that there is a $K > 0$ such that for all $B \subseteq \square^n\mathbb{R}$,

$$w(\square f(B)) \le Kw(B). \tag{41}$$

If we replace (41) by

$$w(\square f(B)) \le Kw(B)^2. \tag{42}$$

then we say $\square f$ has **quadratic convergence**.

Moore originally conjectured that the **centered form** expansion of f has quadratic convergence. The idea of centered forms is to use a Taylor expansion about the midpoint $m(I)$ of I. This was first shown by Hansen and generalized by Nickel and Krawcyk. (SEE STAHL THESIS). Below, we shall reproduce a simple proof by Stahl [32]. INCOMPLETE.

Let us begin with the simplest case, $n = 1$ and where f is a polynomial of degree d. Letting $c = m(I)$, we have

$$f(x) = f(c) + f'(c)(x-c) + \frac{1}{2}f''(c)(x-c)^2 + \cdots + \frac{1}{d!}f^{(d)}(c)(x-c)^d \tag{43}$$

$$= f(c) + \sum_{i=1}^{d} f^{[i]}(c)(x-c)^i \tag{44}$$

where, for simplicity, we write $f^{[i]}(x)$ for $\frac{1}{i!}f^{(i)}(x)$, called the normalized ith derivative of f. We define $\square f(I)$ to be the interval evaluation of the expression in (43), i.e.,

$$\square f(I) = f(c) + f'(c)(I-c) + \frac{1}{2}f''(c)(I-c)^2 + \cdots + \frac{1}{d!}f^{(d)}(c)(I-c)^d. \tag{45}$$

There are two remarks about this definition of $\square f(I)$. First, the Taylor expansion depends on the center of I. Second, the intervals $I - c$ are **centered intervals**, i.e., intervals of the form $[-a, a]$ for some $a \ge 0$. For simplicity, we shall write $[\pm a]$ for $[-a, a]$ if $a \ge 0$, or $[a, -a]$ if $a < 0$.

The interval evaluation of (45) can advantage of such centered intervals in implementations. We have the following elementary properties of interval operations: let $a, b, c \in \mathbb{R}$.

- c $[\pm a] = [\pm|c| \cdot |a|]$
- $[\pm a] = a[\pm 1]$
- $[\pm a] + [\pm b] = [\pm|a| + |b|]$
- $[\pm a] - [\pm b] = [\pm|a| + |b|]$
- $[\pm a] \times [\pm b] = [\pm|a| \cdot |b|]$

Basically, such centered intervals are parametrized by one real number, and we need only one arithmetic operation to carry out any interval arithmetic operation. As consequence of these elementary rules, we see that the sub-distributive law is really an identity:

$$[\pm c]([\pm a] \pm [\pm b]) = [\pm c][\pm a] \pm [\pm b][\pm b]$$

since both sides are equal to

$$|c|(|a| + |b|)[\pm 1].$$

Consider shifts of centered forms, i.e., intervals represented as $I = c \pm [\pm b]$. Of course, this can represent all intervals, but we are interested in basic properties if I in terms of c, b. In particular, we have $m(I) = c$. We will shortly consider division by I, and we need the following property. Clearly,

$$0 \in I \Leftrightarrow |c| \le |b|.$$

Next, assume $|c| > |b|$. Hence $0 \notin I$ and the expression $[\pm a]/(c + [\pm b])$ is well-defined. Then we have

$$\frac{[\pm a]}{c + [\pm b]} = \frac{[\pm a]}{[c - |b|, c + |b|]} = \frac{[\pm a]}{[|c| - |b|]}. \tag{46}$$

It follows that

$$w\left(\frac{[\pm a]}{c + [\pm b]}\right) = \frac{2|a|}{|c| - |b|}. \tag{47}$$

We are now ready to evaluate a polynomial expression $f(x) = \sum_{i=0}^{d} c_i x^i$ at a centered interval $[\pm a]$ where $a > 0$. We have

$$\begin{aligned} \sum_{i=0}^{d} c_i [\pm a]^i &= c_0 + \left(\sum_{i=1}^{d} |c_i| a^i\right)[\pm 1] && (48)\\ &= c_0 + [\pm b] && (49) \end{aligned}$$

where $b = \sum_{i=1}^{d} |c_i| a^i$. Hence the basic algorithm for $\Box f([\pm a])$ goes as follows:
1. Compute all the normalized Taylor coefficients at $c = m(I)$, i.e., $f^{[i]}(c)$ for $i = 0, \ldots, d$.
2. Compute $b = \sum_{i=1}^{d} |f^{[i]}(c)| a^i$ (note the i in the summation begins with $i = 1$).
3. Return $|f(c)| \pm [\pm b]$.

Moreover, it is clear that

$$m(\Box f(I)) = f(c)$$

and

$$w(\Box f(x)) = b = \sum_{i=1}^{d} |f^{[i]}(c)| a^i.$$

Note that we can easily check if $0 \in \Box f(I)$ since this amounts to $|f(c)| \le b$.

¶10. Box Rational Functions. We extend the previous development to rational functions. We begin with the identity when $f(x) = p(x)/q(x)$:

$$f(x) - f(c) = \frac{p(x) - p(c) - f(c)(q(x) - q(c))}{q(x)}. \tag{50}$$

Then by Taylor's expansion,

$$\begin{aligned} f(x) - f(c) &= \frac{\sum_{i\ge 1} p^{[i]}(c)(x-c)^i - f(c)\sum_{i\ge 1} q^{[i]}(c)(x-c)^i}{\sum_{i\ge 0} q^{[i]}(c)(x-c)^i}\\ &= \frac{\sum_{i\ge 1}(p^{[i]}(c) - q^{[i]}(c))(x-c)^i}{\sum_{i\ge 0} q^{[i]}(c)(x-c)^i}\\ &= \frac{\sum_{i\ge 1} t_i (x-c)^i}{\sum_{i\ge 0} q^{[i]}(x-c)^i} \end{aligned}$$

where

$$t_i := p^{[i]}(c) - q^{[i]}(c) \tag{51}$$

This last expression yields our **standard center form** for rational functions:

$$\Box f(I) := f(c) + \frac{\sum_{i\ge 1} t_i (I-c)^i}{\sum_{i\ge 0} q^{[i]}(I-c)^i}. \tag{52}$$

Moreover, if $c = m(I)$, then $I - c$ is a centered interval and thus $\Box f(I)$ can be easily evaluated similar to the polynomial case.

We can generalize the preceding development as follows: for any $k = 1, \ldots, n$, we generalize (**??**) and the subsequent Taylor expansion to obtain:

$$f(x) - \sum_{i=0}^{k-1} f^{[i]}(c)(x-c)^i = \frac{\sum_{i \geq k} t_{k,i}(x-c)^i}{\sum_{i \geq 0} q^{[i]}(x-c)^i} \tag{53}$$

where

$$t_{k,i} = p^{[i]}(c) - \sum_{j=0}^{k-1} \binom{i}{j} f^{[j]}(c) q^{[i-j]}(c). \tag{54}$$

Note that (51) is just the case $k = 1$.

If we replace x by I in the expression (53), we obtain the **kth order centered form** $\square_k f(I)$. Thus (52) corresponds to $k = 1$. As shown in [29, Section 2.4], the higher order centered forms are at least as good than lower order ones in the sense that

$$\square_{k+1} f(I) \subseteq \square_k f(I).$$

For a polynomial f, this inclusion is always an identity: $\square_k f(I) = \square_1 f(I)$. But for non-polynomial f, the inclusion is strict for general I.

We can further generalize the above center forms to multivariate rational functions.

¶11. Krawczyk's Centered Form. The number of arithmetic operations to compute the above centered forms for $f = p/q$ is $\Theta(n^2)$, where $n = \deg(p) + \deg(q)$. Krwaczyk (1983) described another centered form which uses only $\Theta(n)$ arithmetic operations.

Let $f = f(X_1, \ldots, X_m)$ be a rational function and $B \in \square^m \mathbb{R}$ is contained in the domain of f. We call $G \in \square\mathbb{R}$ an **interval slope** of f in B if

$$f(x) - f(c) \subseteq G \cdot (x - c), \qquad \text{for all } x \in B$$

where $c = m(B)$.

We provide a method to compute G from any straightline program S for f. Such a straightline program is a finite sequence of steps. The ith step $(i = 1, 2, \ldots m)$ introduces a brand new variable u_i. Each step is an assignment statement, of one of the following type:

1. $u_i \leftarrow X_j$ $(j = 1, \ldots, m)$
2. $u_i \leftarrow c$ $(c \in \mathbb{R})$
3. $u_i \leftarrow u_j \circ u_k$ $(j < i, k < i$ and $\circ \in \{\pm, \times, \div\})$

So each u_i represents a rational function in $X_1, \ldots, X_n$. We say S computes the rational function represented by u_m, the last variable. We convert S to S' as follows:

- $G_i \leftarrow 1$ if the ith step is $u_i \leftarrow X_j$
- $G_i \leftarrow 0$ if the ith step is $u_i \leftarrow c$
- $G_i \leftarrow G_j \pm G_k$ if the ith step is $u_i \leftarrow u_j \pm u_k$
- $G_i \leftarrow$ ... if the ith step is $u_i \leftarrow u_j \times u_k$
- $G_i \leftarrow$ ... if the ith step is $u_i \leftarrow u_j / u_k$

We now prove that G_m is an interval slope for f.

Exercises

Exercise 7.1: Prove the assertions in Example 1. ♢

Exercise 7.2: Provide details of interval arithmetic under the midpoint representation. ♢

End Exercises

§8. Additional Notes

Until the 1980's, floating point arithmetic are often implemented in software. Hardware implementation of floating point arithmetic requires an additional piece of hardware ("co-processor"), considered an option. Today, floating point processing is so well-established that this co-procesor is accepted as standard computer hardware. The fact that the floating point numbers is now the dominant machine number representation is, in retrospect, somewhat surprising. First note some negative properties of FP computation, as compared to fixed point computation:

(1) Algorithms for FP arithmetic is much more complicated. This fact is very obvious[9] at the hardware level.

(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, 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 anecdotes, examples and lists of numerical pitfalls (sometimes called "abuses") collected from the early days of numerical computing. Most of these issues are still relevant today. (e.g., [34, 31]

Despite all this, FP computation has become the *de facto* standard for computing in scientific and engineering applications. 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. First, let us note that criterion (1) is no longer an critical issue because the algorithms and circuit design methodology for FP arithmetic are well-understood and relatively stable. Also, minimizing hardware size is usually not the bottleneck in today's world of very large scale integrated (VLSI) circuits. But what are some advantages of FP computation? The first advantage is range: for a given budget of bits, the range of numbers which can be approximated by floating point numbers is much larger than, say 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 domains, this advantage is now less importance with advancing computer speed and increasing hardware complexity. The second advantage is speed: the comparison here is between floating point arithmetic with rational arithmetic. Both floating point numbers and rational numbers are dense in the reals, and are thus useful for approximating real computation. The speed of floating point arithmetic is reduced to integer arithmetic plus some small overhead (and integer arithmetic is considered to be fast). In contrast to rational arithmetic is considerably slower than integer arithmetic, and easily suffer from rapid growth in bit lengths.

§9. APPENDIX: Concepts from Numerical Analysis

We recall some additional concepts from in numerical analysis.

Vector Norms. We assume vectors in $x \in \mathbb{C}^n$ (or $\mathbb{R}^n$). In order to measure the "size" of x, perhaps to estimate errors in a computation, we need some formal concept of size. This is captured by the definition of a norm. A **norm** on $\mathbb{C}^n$ is a function $N : \mathbb{C}^n \to \mathbb{R}$ such that for all $x, y \in \mathbb{C}^n$,

- $N(x) \geq 0$, with equality iff $x = 0$.
- $N(\alpha x) = |\alpha| N(x)$ for all $\alpha \in \mathbb{C}$.
- $N(x + y) \leq N(x) + N(y)$

[9]An examination of the physical sizes of computer chips for FP arithmetic units and for integer arithmetic units will show the vast gap in their relative complexity.

The main example of norms are the ***p*-norms** for any positive integer p. This is denoted $N(x) = \|x\|_p$ and defined as

$$\|x\|_p := \sqrt[p]{|x_1|^p + \cdots |x_n|^p}$$

where $x = (x_1, \ldots, x_n)^T$. The special cases of $p = 1$ and $p = 2$ are noteworthy. In particular, the 2-norm is differentiable and invariant under an orthogonal transformation of the space. We can also generalize this to $p = \infty$ and define

$$\|x\|_\infty := \max |x_1|, \ldots, |x_n|.$$

A fundamental fact is that any two norms N and N' are equivalent in the following sense: there exists positive constants $c < C$ such that for all $x \in \mathbb{C}^n$,

$$c \cdot N(x) \leq N'(x) \leq C \cdot N(x).$$

This notion of equivalence is transitive. Hence, it suffices to show that any norm N is equivalent to the ∞-norm. Write $x = \sum_{i=1}^n x_i e^i$ where e^i is the ith elementary vector. Then

$$N(x) \leq \sum_{i=1}^n |x_i| N(e^i) \leq \|x\|_\infty \sum_{i=1}^n N(e^i)$$

so it is sufficient to choose $C = \sum_{i=1}^n N(e^i)$. Now, consider the unit sphere under the ∞-norm, $S = \{x \in \mathbb{C}^n : \|x\|_\infty = 1\}$. Since S is compact, the norm function $N : S \to \mathbb{R}$ achieves its mininum value at some $x^0 \in S$. Let $N(x^0) = c$. If $\|x\|_\infty = b$ then we have

$$N(x) \geq N(b \cdot \frac{x}{b}) = bN(\frac{x}{b}) \geq b \cdot c = c\|x\|_\infty.$$

This completes the proof.

Normwise vs. componentwise norm. $|A| = [|a_i j|]$

Distance to the closest singularity. The numerical stabilty of a numerical problem is directly influenced by its distance to the nearest singularity. We show the following result from Turing (and Banach in the 1920s). It was first shown by Gastinel for arbitrary norms in 1966 [17]. For a non-singular square matrix A, let

$$\delta_T(A) := \inf_S \{ \frac{\|S - A\|}{\|A\|}$$

where S ranges all singular matrices. Thus $\delta_T(A)$ is the relative distance from A to the nearest singular matrix S. The subscript refers to Turing.

Theorem 9 (Turing).

$$\delta_T(A) = \frac{1}{\|A^{-1}\| \|A\|}.$$

Proof. Choose S such that $\|S - A\| < 1/\|A^{-1}\|$. ... **Q.E.D.**

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, New York, 1983. Translated from German by Jon Rokne.

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

[3] D. H. Bailey. Multiprecision translation and execution of Fortran programs. *ACM Trans. on Math. Software*, 19(3):288–319, 1993.

[4] M. Benouamer, D. Michelucci, and B. Péroche. Boundary evaluation using a lazy rational arithmetic. In *Proceedings of the 2nd ACM/IEEE Symposium on Solid Modeling and Applications*, pages 115–126, Montréal, Canada, 1993. ACM Press.

[5] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. on Math. Software*, 4:57–70, 1978.

[6] 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.

[7] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

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

[9] 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.

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

[11] P. Gowland and D. Lester. A survey of exact arithmetic implementations. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 30–47. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

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

[13] 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.

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

[15] T. Hull, M. Cohen, J. Sawchuk, and D. Wortman. Exception handling in scientific computing. *ACM Trans. on Math. Software*, 14(3):201–217, 1988.

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

[17] W. Kahan. Numerical linear algebra. *Canadian Math. Bull.*, 9:757–801, 1966.

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

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

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

[21] S. Krishnan, M. Foskey, T. Culver, J. Keyser, and D. Manocha. PRECISE: Efficient multiprecision evaluation of algebraic roots and predicates for reliable geometric computation. In *17th ACM Symp. on Comp. Geometry*, pages 274–283, 2001.

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

[23] 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.

[24] D. Michelucci and J.-M. Moreau. Lazy arithmetic. *IEEE Transactions on Computers*, 46(9):961–975, 1997.

[25] R. E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[26] N. T. Müller. The iRRAM: Exact arithmetic in C++. In J. Blank, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, pages 222–252. Springer, 2000. 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers, Lecture Notes in Computer Science, No. 2064.

[27] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, 2000. To appear.

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

[29] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Horwood Publishing Limited, Chichester, West Sussex, UK, 1984.

[30] S. M. Rump. Fast and parallel interval arithmetic, 200X.

[31] K. Schittkowski. *Numerical Data Fitting in Dynamical Systems – A Practical Introduction with Applications and Software*. Kluwer Academic Publishers, 2002.

[32] V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. Ph.D. thesis, Johannes Kepler University, Linz, 1995.

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

[34] von Siegfried M. Rump. How reliable are results of computers? *Jahrbuch Überblicke Mathematik*, pages 163–168, 1983. Trans. from German *Wie zuverlässig sind die Ergebnisse unserer Rechenanlagen?*

[35] C. K. Yap. On guaranteed accuracy computation. In F. Chen and D. Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.

[36] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.

[37] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, Singapore, 2nd edition, 1995.