Lecture 4

Number Types I

We will study arbitrary precision integers, rationals, fixed precision floating point numbers, and arbitrary precision floating point numbers. In later lectures, we will learn about algebraic expressions and general algebraic numbers. We start out with a short discussion of arbitrary precision integers and rationals. The bulk of the lecture will be about floating point numbers.

Floating point numbers are of the form

$$s \cdot m \cdot 2^e$$

where s is a *sign bit* (-1 or $+1$), m is a non-negative number called *mantissa* and e is an integer called *exponent*. The number of digits available for the mantissa is either fixed (all hardware floating point systems) or arbitrary (most software floating point systems). The exponent either comes from a fixed range (hardware floating point numbers and some software floating point systems) or is arbitrary (some software floating point systems). Already the first programmable computer offered floating point numbers. In 1938, Konrad Zuse completed the ”Z1”, the first programmable computer. It worked with 22-bit floating-point numbers having a 7-bit exponent, 15-bit significant (including one implicit bit), and a sign bit. The Z3, completed in 1941, implemented floating point arithmetic exceptions with representations for plus and minus infinity and undefined. The first commercial computers offering floating point arithmetic in hardware are Zuse’s Z4 in 1950, followed by the IBM 704 in 1954. The IEEE standard 754-1985 [18] defines single and double precision floating point arithmetic which is implemented in hardware on all modern processors. Floating point arithmetic (hardware and software) is the workhorse for all scientific and geometric computations and therefore we need to study it carefully. The preceding statement concerning the importance of floating point computations seems to contradict the findings of Lecture **??**. It does not. In the preceding lecture, we showed that a naive substitution of floating point arithmetic for real arithmetic does not work. In the course we will learn that the wise use of floating point arithmetic is one of cornerstones of reliable and efficient geometric computing. *We will teach you how to draw reliable conclusions from approximate arithmetic.*

4.1 Built-In Integers and Arbitrary Precision Integers

Hardware and programming languages provide fixed precision integer arithmetic, usually in signed and unsigned form. Let w be the word size of the machine and let $m = 2^w$. Most current workstations have $w = 32$ or $w = 64$. The unsigned integers consist of the integers between 0 and $m-1$ (both inclusive) and arithmetic is modulo m. The signed integers form an interval [`MININT`,`MAXINT`]. On most machines signed integers are represented in two’s complement. Then `MININT` $= -2^{w-1}$ and `MAXINT` $= 2^{w-1}-1$. An arithmetic operation on signed integers may produce a result outside the range of representable numbers; one says that the operation underflows or overflows. The treatment of overflow and underflow is not standardized, in particular, it is not guaranteed that they lead to a runtime error, in fact they usually do not. For example, the addition `MAXINT` + `MAXINT` has result -2 on the KM's machine, since adding $011\ldots1$ to itself yields $11\ldots10$, which is the representation of -2 in two's complement.

Arbitrary integers are readily implemented in software, for example, in packages [15] and [19, Class BigInteger]. The running time of addition and subtraction is linear in the number of digits. All packages implement some form of fast integer multiplication. Depending on the method used, the running time of multiplication is $O(L^{\log 3})$ or $O(L \log L \log\log L)$, where L is the number of digits in the operands.

nces??

Exercise 0.1: The greatest common divisor of two integers x and y with $x \ge y \ge 0$ can be computed by the recursion $\mathtt{GCD}(x,y) = x$ if $y = 0$ and $\mathtt{GCD}(x,y) = \mathtt{GCD}(y, x \mod y)$ if $y > 0$. Prove that the number of recursive calls is at most proportional to the length of y. Hint: Assume $x > y$ and let $x_0 = x$ and $x_1 = y$. For $i > 1$ and $x_{i-1} \neq 0$ let $x_i = x_{i-2} \mod x_{i-1}$. Let $x_k = 0$ be the last element in the sequence just defined. Relate this sequence to the gcd-algorithm. Show that $x_{k-1} > 0$ and $x_{i-2} \ge x_{i-1} + x_i$ for $i < k$. Conclude that x_{k-j} is at least as large as the j-th Fibonacci number. $\diamondsuit$

Exercise 0.2: The standard algorithm for multiplying two L-bit integers has running time $O(L^2)$. Karatsuba's method ([21]) runs in time $O(L^{\log 3})$. In order to multiply two numbers x and y it writes $x = x_1 \cdot 2^{L/2} + x_2$ and $y = y_1 \cdot 2^{L/2} + y_2$, where x_1, x_2, y_1, and y_2 have $L/2$ bits. Then it computes $z = (x_1 + x_2) \cdot (y_1 + y_2)$ and observes that $x \cdot y = x_1 \cdot y_1 \cdot 2^L + (z - x_1 y_1 - x_2 y_2) \cdot 2^{L/2} + x_2 y_2$. In this way only three multiplications of $L/2$-bit integers are needed to multiply two L-bit integers. The standard algorithm requires four. $\diamondsuit$

4.2 Rational Numbers

A rational number is the quotient of two integers. Addition and multiplication of rational numbers are exact. A rational is normalized, if numerator and denominator are relatively prime. Normalization requires to find the greatest common divisor of numerator and denominator and two divisions to remove it. Normalization is fairly costly. However, one should be aware that some algorithms lead to non-normalized numbers and require normalization for efficiency. A prime example is Gaussian elimination. Consider Gaussian elimination of a 3×3 matrix.

$$\begin{aligned}
\begin{pmatrix} a & b & e \\ c & d & f \\ g & h & i \end{pmatrix} &\to \begin{pmatrix} a & b & e \\ 0 & d - b(c/a) & f - e(c/a) \\ 0 & h - b(g/a) & i - e(g/a) \end{pmatrix} \\
&\to \begin{pmatrix} a & b & e \\ 0 & (ad - bc)/a & (af - ec)/a \\ 0 & (ah - bg)/a & (ai - eg)/a \end{pmatrix} \\
&\to \begin{pmatrix} a & b & e \\ 0 & (ad - bc)/a & (af - ec)/a \\ 0 & 0 & (ai - eg)/a - \frac{(ah-bg)/a}{(ad-bc)/a}(af - ec)/a \end{pmatrix}
\end{aligned}$$

We now have a close look at the element in position $(3,3)$. We have:

$$\begin{aligned}(ai-eg)/a - \frac{(ah-bg)/a}{(ad-bc)/a}(af-ec)/a &= \frac{(ai-eg)(ad-bc)-(ah-bg)(af-ec)}{a(ad-bc)}\\ &= \frac{\text{all terms containing } a + (egbc-bgec)}{a(ad-bc)},\end{aligned}$$

i.e., numerator and denominator contain the common factor a. If common factors are not cleared out in Gaussian elimination, the length of the numbers grows exponentially in the dimension of the matrix. If entries are kept in normalized form, Gaussian elimination is polynomial [**?**].

The use of rational arithmetic is inefficient and should be avoided.

4.3 Floating Point Numbers

We start out with a definition of binary floating point systems. We explain the representation of numbers and the key properties of floating point arithmetic. We move on to derive error bounds for the evaluation of expressions. We will use them extensively in the course: for optimized evaluations of geometric predicates in this lecture, as the basis for an efficient linear kernel (Lecture **??**), for the analysis of perturbation techniques (Lecture **??**), as the computational basis for the exact evaluation of algebraic expressions (Lecture **??**) and, more generally, arithmetic with algebraic numbers (Lecture **??**).

Hardware floating point arithmetic is standardized in the IEEE floating point standard [16, 17, 18]. A floating point number is specified by a sign s, a mantissa m, and an exponent e. The sign is $+1$ or -1. The mantissa consists of t bits $m_1, \ldots, m_t$, and e is an integer in the range $[e_{min}, e_{max}]$. The range of possible exponents contains zero and $e_{min} = -\infty$ and/or $e_{max} = +\infty$ is allowed.

TODO: does $e_{min} = -\infty$ really make sense? Then F is dense in $\mathbb{R}$ at 0. Check that all arguments stay valid. TODO

The number represented by the triple (s,m,e) is as follows:

- If $e_{min} < e \le e_{max}$, the number is $s \cdot (1 + \sum_{1\le i\le t} m_i 2^{-i}) \cdot 2^e$. This is called a *normalized* number.

- If $e = e_{min}$ then the number is $s \cdot \sum_{1\le i\le t} m_i 2^{-i} 2^{e_{min}+1}$. This is called a *subnormal* number. Observe that the exponent is $e_{min}+1$. This is to guarantee that the distance of the largest subnormal number $(1-2^{-t})2^{e_{min}+1}$ and the smallest normalized number $12^{e_{min}+1}$ is small.

- In addition, there are the special numbers $-\infty$ and $+\infty$ and a symbol NaN which stands for not-a-number. It is used as an error indicator, e.g., for the result of a division by zero.

Double precision floating point numbers are represented in 64 bits. One bit is used for the sign, 52 bits for the mantissa ($t = 52$) and 11 bits for the exponent. These 11 bits are interpreted as an integer $f \in [0..2^{11}-1] = [0..2047]$. The exponent $e = f - 1023$; $f = 2047$ is used for the special values and hence $e_{min} = -1023$ and $e_{max} = 1023$. The rules for $f = 2047$ are:

- If all m_i are zero and $f = 2047$ then the number is $+\infty$ or $-\infty$ depending on s.

- In $f = 2047$ and some m_i is non-zero, the triple represents NaN (= not a number).

Let $F = F(t, e_{min}, e_{max})$ be the set of real numbers (including $+\infty$ and $-\infty$) that can be represented as above. A number in F is called *representable*, a number in $\mathbb{R} \setminus F$ is called *non-representable*. Observe that for normalized numbers, the leading 1 is not stored. It is sometimes called the hidden bit. The largest positive representable number (except for ∞) is $\mathtt{MAX}_F = (2 - 2^{-t}) \cdot 2^{e_{max}}$, the smallest positive representable number is
???
$\mathtt{MIN}_F = 2^{-t} \cdot 2^{e_{min}+1} = 2^{-t+e_{min}+1}$, and the smallest positive normalized representable number is $\mathtt{MINNORM}_F = 1 \cdot 2^{e_{min}+1} = 2^{e_{min}+1}$. We define the *normal range* of F as

$$[-\mathtt{MAX}_F, -\mathtt{MINNORM}_F] \cup [\mathtt{MINNORM}_F, \mathtt{MAX}_F]$$

??
and the *subnormal range* as the open interval $(-\mathtt{MINNORM}_F, +\mathtt{MINNORM}_F)$. Observe that 0 lies in the subnormal range. The *range* of F is the closed interval $[-\mathtt{MAX}_F, +\mathtt{MAX}_F]$. We require $\mathtt{MINNORM}_F \leq 2{-}t$. This guarantees $\mathtt{MIN}_F^{1/2} \geq \mathtt{MINNORM}_F$.

Exercise 0.3: Specialize the definitions above to double precision floating point numbers. $\diamondsuit$

4.3.1 Rounding

F is a discrete subset of $\mathbb{R}$. For any real x, let[1] $\mathit{flu}(x)$ be the smallest floating point number greater then or equal to x and let $\mathit{fld}(x)$ be the largest floating point number smaller than or equal to x, i.e.,

$$\mathit{flu}(x) = \min\{z \in F \mid x \leq z\} \quad \text{and} \quad \mathit{fld}(x) = \max\{z \in F \mid z \leq x\}.$$

If x is representable, $\mathit{flu}(x) = \mathit{fld}(x) = x$. If $x > \mathtt{MAX}_F$, $\mathit{flu}(x) = +\infty$ and $\mathit{fld}(d) = \mathtt{MAX}_F$, and if $0 \leq x \leq \mathtt{MIN}_F$, $\mathit{flu}(x) = \mathtt{MIN}_F$ and $\mathit{fld}(x) = 0$.

Rounding a real number x yields $\mathit{flu}(x)$ or $\mathit{fld}(x)$. There are several rounding modes: *Rounding away from zero* yields $\mathit{flu}(x)$ for a nonnegative x and $\mathit{fld}(x)$ for a negative x. *Rounding towards zero* yields $\mathit{fld}(x)$ for a nonnegative x and $\mathit{flu}(x)$ for a negative x. *Rounding to nearest* yields $\mathit{flu}(x)$ or $\mathit{fld}(x)$ depending on which number is closer to x. If both numbers are equally close, i.e., $x = (\mathit{flu}(x) + \mathit{fld}(x))/2$, the result of the rounding has an even last bit in the mantissa. The latter rule makes the rounding deterministic; also there is empirical evidence [**?**] that "rounding to even" in the case of ties has superior computational properties. Rounding to nearest is the default rounding mode in the IEEE standard and we follow this convention. We use $\mathrm{fl}(x)$ to denote the result of rounding x to the nearest floating point number. If $x > \mathtt{MAX}_F$, we define $\mathrm{fl}(x) = \infty$, and if $x < -\mathtt{MAX}_F$, we define $\mathrm{fl}(x) = -\infty$. The following theorem states that rounding of numbers in the normal range incurs a small relative error.

THEOREM 1. *If $x \in \mathbb{R}$ lies in the normal range,*

$$\max(|x - \mathit{flu}(x)|, |x - \mathit{fld}(d)|) \leq 2^{-t} \min(|x|, |\mathit{fld}(x)|, |\mathit{flu}(x)|) \tag{1}$$

and

$$|x - \mathit{fl}(x)| \leq 2^{-t-1} \min(|x|, |\mathit{fl}(x)|). \tag{2}$$

If $|x| > \mathtt{MAX}_F$, $|x - \mathit{fl}(x)| \leq 2^{-t-1} |\mathit{fl}(x)|$.

[1] *flu* stands for "float-up" and *fld* stands for "float-down".

Proof. We may assume that x is positive. Then $\mathtt{MINNORM}_F \le x \le \mathtt{MAX}_F$ and hence $x = m2^e$ for some m and e with $1 \le m < 2$ and $e_{min} \le e \le e_{max}$. If $e = e_{max}$, we have in addition $m \le 2 - 2^{-t}$. The distance between adjacent floating point numbers with exponent e is 2^{-t+e}. Also, $\min(|x|, |fld(x)|, |flu(x)|) \ge 2^e$. Thus

$$\max(|x - flu(x)|, |x - fld(d)|) \le 2^{-t+e} \le 2^{-t}\min(|x|, |fld(x)|, |flu(x)|).$$

The second claim follows from $|x - \mathrm{fl}(x)| \le 2^{-t-1+e}$. Finally, if $|x| > \mathtt{MAX}_F$, $|\mathrm{fl}(x)| = \infty$ and this implies the third claim. □

For subnormal numbers, the relative error of rounding may be arbitrarily large. For example for, $x = \mathtt{MIN}_F/2$ we have $\mathrm{fl}(x) = 0$ and hence $|\mathrm{fl}(x) - x| = x$. Relative to x, the error is 1, and relative to $\mathrm{fl}(x)$, the error in $+\infty$. However, the absolute error is bounded.

LEMMA 2. *Let $x \in \mathbb{R}$ be in the subnormal range. Then*

$$|x - fl(x)| \le 2^{-t-1+e_{min}+1} = 2^{-t-1}\mathtt{MINNORM}_F.$$

Proof. The distance between subnormal floating point numbers is $2^{-t+e_{min}+1}$. □

The quantities 2^{-t} and 2^{-t-1} are so important that they deserve a name. We call $\varepsilon = 2^{-t}$ the *precision* of the floating point system and $\mathbf{u} = 2^{-t-1}$ the *unit of roundoff*.

THEOREM 3 (Quality of Rounding Function). *For any real x,*

$$|x - fl(x)| \le \mathbf{u}\max(|fl(x)|, \mathtt{MINNORM}_F) \tag{3}$$

4.3.2 Arithmetic on Floating Point Numbers

Arithmetic on floating point numbers is only approximate; it incurs roundoff error. Although floating point arithmetic is inherently inexact, the IEEE standard guarantees that the result of any arithmetic operation is close to the exact result, frequently as close as possible. It is important to distinguish between mathematical operations and their floating point implementations. We use $\oplus$, $\ominus$, $\odot$, and $\oslash$ for the floating point implementations of addition, subtraction, multiplication and division, respectively. Only in this lecture, we use $^{1/2}$ for the square-root operation and $\sqrt{\ }$ for its floating point implementation. Generally, we use $\tilde{\circ}$ for the floating point implementation of $\circ$. *The floating point implementations of the operations $+$, $-$, $\cdot$, $/$, and $^{1/2}$ yield the best possible result.* This is an axiom of floating point arithmetic.

DEFINITION 1. *If $x,y \in F$ and $\circ \in \{+,-,\cdot,/\}$ then*

$$x \,\tilde{\circ}\, y = fl(x \circ y)$$

and

$$\sqrt{x} = fl(x^{1/2}).$$

As an immediate consequence of this definition and Theorem 3 we obtain:

THEOREM 4 (Error Bound for Single Operations). *If $x,y \in F$ and $\circ \in \{+,-,\cdot,/\}$ then*

$$|x \tilde{\circ} y - x \circ y| \le \mathbf{u} \max(|x \tilde{\circ} y|, \texttt{MINNORM}_F) \tag{4}$$
$$|x \circ y| \le (1+\mathbf{u}) \max(|x \tilde{\circ} y|, \texttt{MINNORM}_F) \tag{5}$$
$$\left|\sqrt{x} - x^{1/2}\right| \le \mathbf{u} \min(x^{1/2}, \sqrt{x}). \tag{6}$$
$$x^{1/2} \le (1+\mathbf{u})\sqrt{x}. \tag{7}$$
$$\sqrt{x} \le (1+\mathbf{u})x^{1/2}. \tag{8}$$

Proof. Inequality (4) follows immediately from Theorem 3 and inequality (5) is a short calculation.

$$|x \circ y| \le |x \circ y - x \tilde{\circ} y| + |x \tilde{\circ} y| \le \mathbf{u} \max(|x \tilde{\circ} y|, \texttt{MINNORM}_F) + |x \tilde{\circ} y| \le (1+\mathbf{u}) \max(|x \tilde{\circ} y|, \texttt{MINNORM}_F).$$

Inequality (6) certainly holds if $x = 0$ and hence $x^{1/2} = \sqrt{x} = 0$ or if $x = +\infty$ and hence $x^{1/2} = \sqrt{x} = \infty$. If $x > 0$, and hence $x \ge MINF$, we have $x^{1/2} \ge \texttt{MINNORM}_F$ and hence $\sqrt{x} \ge \texttt{MINNORM}_F$. Inequality (6) then follows from (2). Inequalities (7) and (8) are immediate consequences of (6). $\square$

Observe that the floating point operations $\oplus$, $\ominus$, $\odot$, $\oslash$ and $\surd$ must return the exact result if this is representable. This is too much to ask for more complex operation, for example logarithms or exponentials. There one requires that the implementation either returns the exact result (if representable) or one of the two adjacent floating point numbers.

We will also need the following properties.

(a) Floating point arithmetic is monotone, i.e., if $a_1 \le a_2$ and $b_1 \le b_2$ then $a_1 \oplus a_2 \le b_1 \oplus b_2$ and if $0 \le a_1 \le a_2$ and $0 \le b_1 \le b_2$ then $a_1 \odot a_2 \le b_1 \odot b_2$.

(b) Multiplication by a power of two incurs no roundoff error, i.e., if $a \in F$ is a power of two, $b \in F$ and $2a$ and ab are in the range of F, then $a \oplus a = 2 \cdot a$ and $a \odot b = a \cdot b$.

(c) If $a+b$ is representable, then $a \oplus b = a+b$ and if ab is representable $a \odot b = ab$.

(d) If $x \in \mathbb{N}$, $x < 2^{t+1}$ and $t \le e \le e_{max}$, then $x2^e$ is representable.

The IEEE standard also defines the results for "strange" combinations of arguments. Of course, division by zero yields NaN. Also, if one of the arguments of an addition is NaN or the addition has no defined result, e.g., $-\infty + \infty$, then the result is NaN.

Exercise 0.4: Let $a,b \in F$ with $\frac{1}{2} \le \frac{a}{b} \le 2$. Show that $a \ominus b = a - b$. This was first observed by Sterbenz [27]. $\diamondsuit$

Exercise 0.5: Assume for this exercise that point coordinates are doubles in $[1/2,1]$. Show

- Orientation$(p,q,r) = 0$ implies *float_orient*$(p,q,r) = 0$.
- *float_orient*$(p,q,r) \ne 0$ implies Orientation$(p,q,r) =$ *float_orient*(p,q,r).
- What does this mean for a figure such as Figure **??**?
- Can you find examples as in Section **??** when point coordinates are restricted to doubles in $[1/2,1]$?

$\diamondsuit$

4.3.3 Floating Point Integers

We briefly discuss the use of double precision hardware floating point arithmetic for 53-bit integer arithmetic. Let us call an integer a *floating point integer* if it belongs to the interval $I := [-(2^{53}-1)..2^{53}-1]$. The numbers in I can be represented as double precision floating point numbers. Consider a non-negative integer $x = \sum_{0\le i\le 53} x_i 2^i \in I$. If $x=0$, x is a double. If $x>0$, let j be maximal such that $x_j \neq 0$. Then $x = (1+\sum_{1\le i\le j} x_{j-i}2^{-i})2^j$ and hence x is a double. Double precision floating point arithmetic on numbers in I is exact.

LEMMA 5. *Assume $x \in I$, $y \in I$ and $x \circ y \in I$ where $\circ \in \{+,-,\cdot\}$. Then $x \tilde{\circ} y = x \circ y$.*

Lemma 5 is useful if points have integer Cartesian or homogeneous coordinates of bounded size.

LEMMA 6. *Assume that points have integral Cartesian coordinates. Then*

$$(b_x - a_x)\cdot(c_y-a_y) - (b_y-a_y)\cdot(c_x-a_x)$$

is computed without roundoff error if the absolute value of all coordinates is bounded by 2^L-1, where $2(L+1)+1 \le 53$.

Proof. The absolute value of the expression is strictly bounded by

$$(2^L+2^L)\cdot(2^L+2^L)+(2^L+2^L)\cdot(2^L+2^L) = 2^{2L+3}.$$

Thus if $2L+3 \le 53$, the value is in I and hence computed correctly. $\square$

Exercise 0.6: Prove an analogous lemma for the orientation predicate and points with integer homogeneous coordinates and for the side-of-circle predicate and points with integer Cartesian or homogeneous coordinates. $\diamondsuit$

Built-in 32-bit integer arithmetic can only handle integers whose absolute value is bounded by $2^{31}-1$. So it supports the orientation predicate for integer coordinates with at most 14 bits. In contrast, doubles support the orientation predicate for integer coordinates with up to 25 bits. One may paraphrase this observation as *doubles are the better ints*.

4.4 An Optimized Evaluation Order for the Orientation Predicate

TODO, Chee's note are a good source.

4.5 An Error Analysis for Arithmetic Expressions

We study the evaluation of simple arithmetic operations in floating point arithmetic. Any real is an arithmetic expression and if A and B are arithmetic expression, then are $A+B$, $A-B$, $A\cdot B$, and $A^{1/2}$. The latter assumes that the value of A is non-negative. For an arithmetic expression E, let $\tilde{E}$ the result of evaluating E with floating point arithmetic. We want to bound

$$\left|\tilde{E}-E\right|.$$

E	condition	$\tilde{E}$	m_E	d_E
a	a is non-representable	$\mathrm{fl}(a)$	$\max(\mathtt{MINNORM}_F, \lvert\mathrm{fl}(a)\rvert)$	1
a	a is representable	a	$\max(\mathtt{MINNORM}_F, \lvert a\rvert)$	0
$A+B$		$\tilde{A} \oplus \tilde{B}$	$m_A \oplus m_B$	$1+\max(d_A, d_B)$
$A-B$		$\tilde{A} \ominus \tilde{B}$	$m_A \oplus m_B$	$1+\max(d_A, d_B)$
$A \cdot B$		$\tilde{A} \odot \tilde{B}$	$\max(\mathtt{MINNORM}_F, m_A \odot m_B)$	$1+d_A+d_B$
$A^{1/2}$	$\tilde{A} < \mathbf{u} m_A$	0	$2^{(t+1)/2}\sqrt{m_A}$	$2+d_A$
$A^{1/2}$	$\tilde{A} \geq \mathbf{u} m_A$	$\sqrt{\tilde{A}}$	$\max(\sqrt{\tilde{A}}, m_A \oslash \sqrt{\tilde{A}})$	$2+d_A$

Table 4.1: The recursive definition of m_E and ind_E. The first column contains the case distinction according to the syntactic structure of E, the second column contains the rule for computing $\tilde{E}$ and the third and fourth columns contain the rules for computing m_E and ind_E; $\oplus$, $\ominus$, $\odot$, and $\oslash$ denote the floating point implementations of addition, subtraction, and multiplication, and $\sqrt{\ }$ denotes the floating point implementation of the square-root operation. Observe that $m_E = \infty$ if either $m_A = \infty$ or $m_B = \infty$.

Such a bound can be used to draw a reliable conclusion about the sign of an expression, because

$$\left|\tilde{E} - E\right| \leq B \quad \text{and} \quad \left|\tilde{E}\right| > B \quad \text{implies} \quad \mathrm{sign}(E) = \mathrm{sign}(\tilde{E}).$$

This observation is very important. It shows that we may be able to determine the sign of an expression with floating point arithmetic although it might be impossible to determine its value with floating point arithmetic.
We will derive a bound of the form

$$\left|E - \tilde{E}\right| \leq B \quad \text{where} \quad B = ((1+\mathbf{u})^{d_E} - 1) \cdot m_E \leq (d_E + 2) \odot \mathbf{u} \odot m_E,$$

and d_E and m_E are defined in Table 4.1. The intuitive interpretation is as follows: m_E upper bounds $\tilde{E}$ and d_E measures the levels of rounding. The operators $+$, $-$, and $\cdot$ introduce one additional level of rounding, the square-root-operator accounts for two levels. In an addition, the arguments contribute the maximum of their levels, and in a multiplication, the arguments contribute their sum. Each level of rounding increases the range of uncertainty by a multiplicative factor of $1+\varepsilon$. The subtraction of a -1 reflects the fact that we are interested in the error.

Before we establish the error bound, we will show that $((1+\mathbf{u})^d - 1)$ is approximately equal to $d\mathbf{u}$ and we will also give an example.

LEMMA 7. *If $d \leq \sqrt{1/\mathbf{u}} - 1$ then $((1+\mathbf{u})^d - 1) \leq (d+1)\mathbf{u}$. For all d, $((1+\mathbf{u})^d - 1) \geq d\mathbf{u}$.*

Proof. We have

$$(1+\mathbf{u})^d - 1 = \sum_{1 \leq i \leq d} \binom{d}{i} \mathbf{u}^i \leq \sum_{i \geq 1} (d \cdot \mathbf{u})^i = d\mathbf{u}/(1 - d\mathbf{u}).$$

Next observe that $d\mathbf{u}/(1-d\mathbf{u}) \leq (1+d)\mathbf{u}$ iff $d/(1-d\mathbf{u}) \leq (1+d)$ iff $d \leq d+1-d^2\mathbf{u} - d\mathbf{u}$ iff $d(d+1) \leq 1/\mathbf{u}$. This is certainly the case when $(d+1)^2 \leq \mathbf{u}$ or $d \leq \sqrt{1/\mathbf{u}} - 1$. The lower bound follows immediately from the expansion of $(1+\mathbf{u})^d$. $\square$

The condition $d \leq \sqrt{1/\mathbf{u}} - 1$ is hardly constraining. For $\mathbf{u} = 2^{-53}$, it amounts to $d < 2^{26.5}$. As an example, we use the orientation predicate for points a, b, and c given by their Cartesian coordinates. Then

$$\mathsf{Orientation}(a,b,c) = (b_x - a_x) \cdot (c_y - a_y) - (b_y - a_y) \cdot (c_x - a_x).$$

We compute the d-value of this expression. The degree of any argument is one, the degree of $(b_x - a_x)$ is 2, the degree of $(b_x - a_x) \cdot (c_y - a_y)$ is 5 and the degree of the entire expression is 6. We conclude that the error of evaluating $\mathsf{Orientation}(a,b,c)$ with floating point arithmetic is at most

$$7 \cdot \mathbf{u} \cdot m_{\mathsf{Orientation}(a,b,c)}.$$

This bound is worth to be formulated as a Lemma.

LEMMA 8. *If points are given by their Cartesian coordinates and the orientation predicate is computed by the formula above, the roundoff error in a floating point evaluation is bounded by* $7 \cdot \mathbf{u} \cdot m_{\mathsf{Orientation}(a,b,c)}$ *(*$8 \odot \mathbf{u} \odot m_{\mathsf{Orientation}(a,b,c)}$*).*

Lemma 8 leads to the following code for evaluation of the orientation predicate. We assume that the Cartesian coordinates belong to some number type *NT* for which we have exact arithmetic available. We first convert all coordinates to a floating point number and then evaluate the orientation precision with floating point arithmetic. If the absolute value of the floating point result is sufficiently big, we return its result. If it is too small we resort to exact computation.

```
int orientation(point_2d p, point_2d q, point_2d r){
NT px = p.xcoord(), py = p.ycoord(), qx = q.xcoord(), .... ;
// evaluation in floating point arithmetic
float pxd = fl(px), pyd = fl(py), qxd = fl(qx), .....;
float Etilde = (qxd - pxd)*(ryd - pyd) - (qyd - pyd)*(rxd - pxd);
float apxd = abs(pxd), apyd = abs(pyd), aqxd = abs(qxd), ....;
float mes = (aqxd + apxd)*(aryd + apyd) + (aqyd + apyd)*(arxd + apxd);
if ( abs(Etilde) > 8 * uu * mes ) return (sign Etilde);
// exact evaluation
NT E = (qx - px)*(ry - py) - (qy - py)*(rx - px);
return sign E;
}
```

Exercise 0.7: Assume that a point p is given by its homogeneous coordinates (px, py, pw). Assuming $\text{sign}(aw \cdot bw \cdot cw) = 1$, we have

$$\mathsf{Orientation}(a,b,c) = aw \cdot (bx \cdot cy - by \cdot cx) - bw \cdot (ax \cdot cy - ay \cdot cx) + cw \cdot (ax \cdot by - ay \cdot bx).$$

Compute the d-value of this expression. $\diamondsuit$

Exercise 0.8: Assume that for i, $1 \leq i \leq 8$, x_i is an integer with $|x_i| \leq 2^{20}$. Evaluate the expression $((x_1 + x_2) \cdot (x_3 + x_4)) \cdot x_5 + (x_6 + x_7) \cdot x_8$ with double precision floating point arithmetic. Derive a bound for the maximal difference between the exact result and the computed result. $\diamondsuit$

THEOREM 9 (Error Bound for Arithmetic Expressions). *If m_E and d_E are computed according to Table 4.1 then*

$$m_E \geq \texttt{MINNORM}_F \quad \textit{and} \quad m_E \geq \left|\tilde{E}\right| \quad \textit{and} \quad \left|\tilde{E} - E\right| \leq ((1+\mathbf{u})^{d_E} - 1) \cdot m_E$$

Proof. We use induction on the structure of the expression E. The claims $m_E \ge \texttt{MINNORM}_F$ and $m_E \ge \left|\tilde{E}\right|$ follow immediately from the table and the monotonicity of floating point arithmetic. For the third claim we have to work harder. We use induction on the structure of E and start by observing that the claim is obvious if $m_E = \infty$. The base case is obvious. If $E = a$ and a is representable, $\tilde{E} = E$. If a is non-representable we invoke Theorem 3.

For the induction step we make a case distinction according to the operation combining A and B. Assume first that $E = A + B$. Then

$$\left|\tilde{E} - E\right| = \left|\tilde{A} \oplus \tilde{B} - (A+B)\right| \le \left|\tilde{A} \oplus \tilde{B} - (\tilde{A}+\tilde{B})\right| + \left|\tilde{A} - A\right| + \left|\tilde{B} - B\right|.$$

Inequality (4) bounds the first term by $\mathbf{u}\max(\left|\tilde{A} \oplus \tilde{B}\right|, \texttt{MINNORM}_F)$. Next observe that

$$\max(\left|\tilde{A} \oplus \tilde{B}\right|, \texttt{MINNORM}_F) \le \max(m_A \oplus m_B, \texttt{MINNORM}_F) = \max(m_E, \texttt{MINNORM}_F) = m_E$$

by monotonicity of floating point arithmetic and since $m_E \ge \texttt{MINNORM}_F$. For the other two terms we use the induction hypothesis to conclude

$$\begin{aligned}
\left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| &\le ((1+\mathbf{u})^{d_A} - 1) \cdot m_A + ((1+\mathbf{u})^{d_B} - 1) \cdot m_B \\
&\le ((1+\mathbf{u})^{\max(d_A,d_B)} - 1) \cdot (m_A + m_B) \\
&\le ((1+\mathbf{u})^{\max(d_A,d_B)} - 1) \cdot (1+\mathbf{u}) \cdot m_E \qquad \text{by inequality (5).}
\end{aligned}$$

Putting the two bounds together we obtain:

$$\begin{aligned}
\left|\tilde{E} - E\right| &\le [\mathbf{u} + ((1+\mathbf{u})^{\max(d_A,d_B)} - 1) \cdot (1+\mathbf{u})] \cdot m_E \\
&= [(1+\mathbf{u})^{1+\max(d_A,d_B)} - 1] \cdot m_E.
\end{aligned}$$

Subtractions are treated completely analogously.

We turn to multiplications, $E = A \cdot B$. We have

$$\left|\tilde{E} - E\right| = \left|\tilde{A} \odot \tilde{B} - A \cdot B\right| \le \left|\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B}\right| + \left|\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}\right| + \left|A \cdot \tilde{B} - A \cdot B\right|.$$

Inequality (4) and monotonicity of floating point arithmetic bound the first term by

$$\mathbf{u}\max(\left|\tilde{A} \odot \tilde{B}\right|, \texttt{MINNORM}_F) \le \mathbf{u}\max(m_A \odot m_B, \texttt{MINNORM}_F) = \mathbf{u} m_E.$$

For the second term we use the induction hypothesis to conclude

$$\begin{aligned}
\left|\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}\right| &= \left|\tilde{A} - A\right| \cdot \left|\tilde{B}\right| \\
&\le ((1+\mathbf{u})^{d_A} - 1) \cdot m_A \cdot m_B \\
&\le ((1+\mathbf{u})^{d_A} - 1) \cdot (1+\mathbf{u}) \cdot \max(m_A \odot m_B, \texttt{MINNORM}_F) \qquad \text{by inequality (5)} \\
&= ((1+\mathbf{u})^{d_A} - 1) \cdot (1+\mathbf{u}) \cdot m_E,
\end{aligned}$$

and for the third term we conclude similarly

$$\begin{aligned}
\left|A \cdot \tilde{B} - A \cdot B\right| &= |A| \cdot \left|\tilde{B} - B\right| \\
&\le (\left|\tilde{A}\right| + \left|\tilde{A} - A\right|) \cdot \left|\tilde{B} - B\right| \\
&\le (1+\mathbf{u})^{d_A} \cdot m_A \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot m_B \\
&\le (1+\mathbf{u})^{1+d_A} \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot \max(m_A \odot m_B, \texttt{MINNORM}_F) \qquad \text{by inequality (5)} \\
&= (1+\mathbf{u})^{1+d_A} \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot m_E
\end{aligned}$$

Putting the three bounds together, we obtain

$$\begin{aligned}\left|\tilde{E}-E\right| &\le (\mathbf{u}+(1+\mathbf{u})\cdot((1+\mathbf{u})^{d_A}-1)+(1+\mathbf{u})^{1+d_A}\cdot((1+\mathbf{u})^{d_B}-1))m_E\\ &= (\mathbf{u}+(1+\mathbf{u})^{d_A+1}-1-\mathbf{u}+(1+\mathbf{u})^{1+d_A+d_B}-(1+\mathbf{u})^{1+d_A})m_E\\ &= ((1+\mathbf{u})^{1+d_A+d_B}-1)m_E\end{aligned}$$

and the induction step is completed for the case of multiplications.

We finally come to square roots, $E=A^{1/2}$. We distinguish cases according to the relative size of $\tilde{A}$ and m_A. Assume first that $\tilde{A}$ is tiny compared to m_A, formally, $\tilde{A}<\mathbf{u}\cdot m_A$. We set $\tilde{E}=0$. Then

$$\begin{aligned}\left|\tilde{E}-A^{1/2}\right| &= \left|A^{1/2}\right|\\ &\le (|\tilde{A}|+|\tilde{A}-A|)^{1/2}\\ &\le (\mathbf{u}\cdot m_A+((1+\mathbf{u})^{d_A}-1)\cdot m_A)^{1/2}\\ &\le (\mathbf{u}+((1+\mathbf{u})^{d_A}-1))^{1/2}(1+\mathbf{u})\cdot\sqrt{m_A} && \text{by inequality (7)}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)\cdot\sqrt{m_A}\mathbf{u}^{-1/2},\end{aligned}$$

where the last inequality uses

$$\begin{aligned}(\mathbf{u}+((1+\mathbf{u})^{d_A}-1))^{1/2}(1+\mathbf{u}) &= [(\mathbf{u}+((1+\mathbf{u})^{d_A}-1))(1+\mathbf{u})^2]^{1/2}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)^{1/2}\\ &\le (\mathbf{u}(d_A+3))^{1/2}\\ &\le (\mathbf{u}(d_A+2))\mathbf{u}^{-1/2}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)\mathbf{u}^{-1/2}.\end{aligned}$$

Assume next that $\tilde{A}\ge\mathbf{u}\cdot m_A$. Then

$$\begin{aligned}\left|\sqrt{\tilde{A}}-A^{1/2}\right| &\le \left|\sqrt{\tilde{A}}-\tilde{A}^{1/2}\right|+\left|\tilde{A}^{1/2}-A^{1/2}\right|\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+\frac{|\tilde{A}-A|}{\tilde{A}^{1/2}+A^{1/2}} && \text{by inequality (6)}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+\frac{((1+\mathbf{u})^{d_A}-1)\cdot m_A}{\tilde{A}^{1/2}}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})\cdot\frac{m_A}{\sqrt{\tilde{A}}} && \text{by inequality (8)}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})^2\cdot\max(m_A\oslash\sqrt{\tilde{A}},\mathtt{MINNORM}_F) && \text{by inequality (5)}\\ &\le (\mathbf{u}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})^2)\cdot\max(m_A\oslash\sqrt{\tilde{A}},\sqrt{\tilde{A}},\mathtt{MINNORM}_F)\\ &= ((1+\mathbf{u})^{d_A+2}-1)\cdot\max(m_A\oslash\sqrt{\tilde{A}},\sqrt{\tilde{A}}),\end{aligned}$$

where the last inequality follows from $\tilde{A}>0$ and hence $\sqrt{\tilde{A}}\ge\mathtt{MINNORM}_F$. This completes the induction step for the case of square roots. □

THEOREM 10. *If* $d_E\le\sqrt{1/\mathbf{u}}-1$ *then*

$$\left|E-\tilde{E}\right|\le(d_E+1)\cdot\mathbf{u}\cdot m_E\le(d_E+2)\odot m_E\odot\mathbf{u}.$$

X	$\tilde{X}$	c_X	k_X	d_X
a	$\mathrm{fl}(a)$	1	1	1
$A+B$	$\tilde{A}\oplus\tilde{B}$	c_A+c_B	$\max(k_A,k_B)$	$1+\max(d_A,d_B)$
$A-B$	$\tilde{A}\ominus\tilde{B}$	c_A+c_B	$\max(k_A,k_B)$	$1+\max(d_A,d_B)$
$A\cdot B$	$\tilde{A}\odot\tilde{B}$	c_Ac_B	k_A+k_B	$1+d_A+d_B$

Table 4.2: The recursive definition of c_X, k_X and d_X. The first column contains the case distinction according to the syntactic structure of X, the second column contains the rule for computing $\tilde{X}$ and the third to fifth columns contain the rules for computing c_X, k_X, and d_X.

Proof. Follows immediately from Theorem 9 and Lemma 7. □

Exercise 0.9: Consider the computation of m_E according to Table 4.1. Show that the rule for square roots cannot lead to overflow (if $e_{max} > t+1$). Give examples, where the rules for addition, subtraction, and multiplication overflow.

Answer: We have $m_A < (2-1/2^t)2^{e_{max}}$. There are two rules for computing $E = m_{A^{1/2}}$. If $\tilde{A} < \mathbf{u}m_A$, we define $m_E = 2^{(t+1)/2} \odot \sqrt{m_A}$. The square-root operation cannot overflow; if the multiplication overflows we certainly have $\sqrt{m_A} > 2^{e_{max}-(t+1)/2}$ or $m_A > 2^{2e_{max}-(t+1)} > 2^{e_{max}}$, a contradiction. If $\tilde{A} \geq \mathbf{u}m_A$, we define $m_E = \max(\sqrt{\tilde{A}}, m_A \oslash \sqrt{\tilde{A}})$. Since $\tilde{A} \leq m_A$, the computation of $\sqrt{\tilde{A}}$ cannot overflow. Also, since $\tilde{A} \geq \mathbf{u}m_A$, $\sqrt{\tilde{A}} \geq \mathbf{u}^{1/2}\sqrt{m_A}$ and hence

$$m_A \oslash \sqrt{\tilde{A}}) \leq m_A \oslash \mathbf{u}^{1/2}\sqrt{m_A} \leq 2^{(t+1)/2}(1+\mathbf{u})^3\sqrt{m_A}$$

and we already shown that the latter quantity does not overflow. ♢

4.6 A Simplified Error Analysis for Polynomial Expressions

The error bounds of the preceding section are for machine consumption and not for human consumption. They should be used to filter the evaluation of geometric predicates. For the analysis of perturbation methods in Lecture **??** a weaker and simpler bound suffices. We next derive such a bound for polynomial expressions, i.e., expressions using only additions, subtractions, and multiplications. We show that

$$\left|\tilde{E} - E\right| \leq ((1+u)^{d_E} - 1)c_E M^{k_E},$$

where d_E, c_E and k_E are defined as in Table 4.2 and M is the smallest power of two such that

$$M \geq \max(1, \max\{\mathrm{lf}(|a|) \mid a \text{ is an operand in } E\}).$$

Exercise 0.10: Prove $M \geq \mathit{flu}(|a|)$ for all operands a in E. ♢

THEOREM 11. *Let M be defined as above. Then for every subexpression X of E,*

$$c_X \geq 1 \quad \textit{and} \quad k_X \geq 0 \quad \textit{and} \quad \left|\tilde{X} - X\right| \leq ((1+\mathbf{u})^{d_X} - 1)c_X M^{k_X},$$

where c_X, k_X and d_X are defined as in Table 4.2. This assumes that $c_X M^{k_X}$ is representable[2] *for all X. The latter assumption also guarantees that the computation of no m_X overflows.*

Proof. We use structural induction. Observe that the rules for d_X are the same as in Theorem 9. It therefore suffices to prove

$$m_X \le c_X M^{k_X}$$

for all X. This is clear for operands. If $X = a \in \mathbb{R}$, $m_X = \max(\mathtt{MINNORM}_F, \mathrm{fl}(a)) \le M$. Consider an addition or subtraction next. Then

$$m_X = m_A \oplus m_B \le c_A M^{k_A} \oplus c_B M^{k_B} \le c_a M^{k_X} \oplus c_B M^{k_X} = (c_A + c_B) M^{k_X} = c_X M^{k_X},$$

where the next to last equality follows from the assumption that $c_X M^{k_X}$ is representable. Finally, we come to a multiplication. If $m_X = \mathtt{MINNORM}_F$, the claim is obvious since $M \ge 1$, $k_X \ge 0$ and $c_X \ge 1$. So assume $m_X = m_A \odot m_B$. Then

$$m_X = m_A \odot m_B \le c_A M^{k_A} \odot c_B M^{k_B} = (c_A c_B) M^{k_X} = c_X m^{k_X},$$

where again the next to last equality follows from our assumption that $c_X M^{k_X}$ is representable.

Finally, since $0 \le m_X \le c_X M^{k_X}$ and the latter quantity is assumed to be representable, the computation of m_X does not overflow. □

We continue our discussion of the orientation predicate for points a, b, and c given by their Cartesian coordinates. Then

$$\mathsf{Orientation}(a,b,c) = \mathrm{sign}((b_x - a_x)\cdot(c_y - a_y) - (b_y - a_y)\cdot(c_x - a_x)).$$

We already determined the degree of this expression as 6. The c- and k-values are as follows. For any argument, both values are one, for $X = b_x - a_x$, we have $c_X = 2$ and $k_X = 1$, for $X = (b_x - a_x)\cdot(c_y - a_y)$, we have $c_X = 4$ and $k_X = 2$, and finally for the entire expression we have $c_X = 8$ and $k_X = 2$. We conclude that the roundoff error in evaluating $\mathsf{Orientation}(a,b,c)$ with floating point arithmetic is at most

$$7 \cdot \mathbf{u} \cdot 8 \cdot M^2 = 56 \cdot \mathbf{u} \cdot M^2.$$

where M is the smallest non-negative power of two bounding all Cartesian coordinates. In particular, if $M = 2^{10}$ and double precision arithmetic is used, the error is at most $54 \cdot 2^{-53} \cdot 2^{20} \le 2^{-27}$. Next recall that the expression underlying Orientation is twice the signed area of the triangle $\Delta(a,b,c)$. Thus, if coordinates are at most 2^{10} and the (unsigned) area of $\Delta(a,b,c)$ is at least 2^{-26}, then *float_orient*$(a,b,c) = \mathsf{Orientation}(a,b,c)$. So *float_orient* errs only for very skinny triangles. Figure **??** suggested this, but now we know for sure. We will exploit the correctness of *float_orient* for non-skinny triangles in Lecture **??**.

Exercise 0.11: Redo the analysis above for points given by their homogeneous coordinates. We continue our discussion of the orientation predicate for points given by their homogeneous coordinates. Assuming $\mathrm{sign}(aw, bw, cw) = 1$, we have

$$\mathsf{Orientation}(a,b,c) = aw\cdot(bx\cdot cy - by\cdot cx) - bw\cdot(ax\cdot cy - ay\cdot cx) + cw\cdot(ax\cdot by - ay\cdot bx).$$

[2]This is certainly the case if $c_X \le 2^{t+1}$ and $M^{k_X} \le 2^{e_{max}}$.

We already determined the degree of this expression as 8. The c- and k-values are as follows. For any argument, both values are one, for $X = bx \cdot cy$, we have $c_X = 1$ and $k_X = 2$, for $X = (bx \cdot cy - by \cdot cx)$, we have $c_X = 2$ and $k_X = 2$, for $X = aw \cdot (bx \cdot cy - by \cdot cx)$ we have $c_X = 2$ and $k_X = 3$, for $X = aw \cdot (bx \cdot cy - by \cdot cx) - bw \cdot (ax \cdot cy - ay \cdot cx)$ we have $c_X = 4$ and $k_X = 3$, and finally for the entire expression we have $c_X = 6$ and $k_X = 3$. We conclude that the roundoff error in evaluating $\mathsf{Orientation}(p,q,r)$ with floating point arithmetic is at most

$$9 \cdot \mathbf{u} \cdot 6 \cdot M^3 = 54 \cdot \mathbf{u} \cdot M^3.$$

where M is the smallest non-negative power of two bounding the absolute value of all arguments. In particular, if $M = 2^{10}$ and double precision arithmetic is used, the error is at most $54 \cdot 2^{-53} \cdot 2^{30} \le 2^{-17}$. If, we increase mantissa length to 99, the error bound becomes 2^{-64}. $\diamondsuit$

Exercise 0.12: Assume that for i, $1 \le i \le 8$, x_i is an integer with $|x_i| \le 2^{20}$. Evaluate the expression $((x_1 + x_2) \cdot (x_3 + x_4)) \cdot x_5 + (x_6 + x_7) \cdot x_8$ with double precision floating point arithmetic. Derive a bound for the maximal difference between the exact result and the computed result. $\diamondsuit$

Exercise 0.13: Extend Theorem **??** to include square-roots. This requires to extend Table **??** and the proof of the theorem. We do not have a satisfactory answer for this exercise. $\diamondsuit$

4.7 A More Precise Error Analysis*

[[I will probably move this section to the chapter on deciding the sign of algebraic expressions.]]

Consider the expression

$$E = (a+b) - a$$

when $a \gg b$. The error analysis of Section 4.5 assumes that the error in the subtraction may be as large as

$$\mathbf{u} m_E \approx \mathbf{u}(2a+b).$$

However, the actual error is approximately

$$\mathbf{u} \cdot \tilde{E} \approx \mathbf{u} \cdot b,$$

which is much smaller. Can we improve our error analysis? Recall our formulae for estimating the error in additions (subtractions) and multiplications. We use err_E to denote $\tilde{E} - E$. For $E = A + B$, we have

$$\begin{aligned} err_E = \left|\tilde{A} \oplus \tilde{B} - (A+B)\right| &\le \left|\tilde{A} \oplus \tilde{B} - (\tilde{A}+\tilde{B})\right| + \left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| \\ &\le \mathbf{u}\left|\tilde{A} \oplus \tilde{B}\right| + \left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| \qquad \le \mathbf{u} \odot \left|\tilde{E}\right| + err_A + err_B). \end{aligned}$$

and for $E = A \cdot B$, we have

$$\begin{aligned} |err_E| = \left|\tilde{A} \odot \tilde{B} - A \cdot B\right| &= \left|\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B} + \tilde{A} \cdot \tilde{B} - A \cdot \tilde{B} + A \cdot \tilde{B} - A \cdot B\right| \\ &\le \mathbf{u}\left|\tilde{A} \odot \tilde{B}\right| + \left|\tilde{A} - A\right| \cdot \left|\tilde{B}\right| + |A| \left|\tilde{B} - B\right| \\ &\le \mathbf{u}\left|\tilde{E}\right| + |err_A| \cdot \left|\tilde{B}\right| + |A| \cdot |err_B| \end{aligned}$$

These error bounds are more costly to evaluate than the bounds in Section 4.5. We will use them in Chapter **??**.

4.8 Arbitrary Precision Floating Point Numbers

In Section 4.3, we introduced the floating point system $F(t, e_{min}, e_{max})$. Software floating point systems are usually more flexible. They allow the user to change t during the computation, either by setting it to a fixed value at the beginning of the computation or by changing it freely during a computation. For some value, one wants a mantissa length of 1000 bits, and for another value, one wants 2000 bits, and for another value, one wants no rounding [3] Exponents are arbitrary integer, i.e., $e_{min} = -\infty$ and $e_{max} = +\infty$. The systems also support the different rounding modes of the IEEE standard. The mode can either be chosen for the entire computation or for a single operation.

As an example, consider the following LEDA program snippet computing an approximation of Euler's number $e \approx 2.71$. Let m be an integer. Our goal is to compute a bigfloat z such that $|z - e| \le 2^{-m}$. Euler's number is defined as the value of the infinite series $\sum_{n \ge 0} 1/n!$. The simplest strategy to approximate e is to sum a sufficiently large initial fragment of this sum with a sufficiently long mantissa, so as to keep the total effect of the rounding errors under control. Assume that we compute the sum of the first n_0 terms with a mantissa length of t bits for still to be determined values of n_0 and t, i.e., we execute the following program.

```
bigfloat::set_rounding_mode(TOZERO);
bigfloat::set_precision(t);
bigfloat z = 2; integer fac = 2; int n = 2;
while (n < n0)
  { // fac = n! and z approximates 1/0! + ... + 1/(n-1)!
    z = z + 1/bigfloat(fac);
    n++; fac = fac * n;
  }
```

Let z_0 be the final value of z. Then z_0 is the value of $\sum_{n<n_0} 1/n!$ computed with bigfloat arithmetic with a mantissa length of t binary places. We have incurred two kinds of errors in this computation: a truncation error since we summed only an initial segment of an infinite series and a rounding error since we used floating point arithmetic to sum the initial segment. Thus,

$$|e - z_0| \le \left| e - \sum_{n<n_0} 1/n! \right| + \left| \sum_{n<n_0} 1/n! - z_0 \right|$$

$$= \sum_{n \ge n_0} 1/n! + \left| \sum_{n<n_0} 1/n! - z_0 \right|$$

The first term is certainly bounded by $2/n_0!$ since, for all $n \ge n_0$, $n! = n_0! \cdot (n_0 + 1) \cdot \ldots \cdot n \ge n_0! \cdot 2^{n-n_0}$ and hence $\sum_{n \ge n_0} 1/n! \le 1/n_0! \cdot (1 + 1/2 + 1/4 + \ldots) \le 2/n_0!$. What can we say about the total rounding error? We observe that we use one floating point division and one floating point addition per iteration and that there are $n_0 - 2$ iterations. Also, since we set the rounding mode to rounding-to-zero, the value of z always stays below e and hence stays bounded by 3. Thus, the results of all bigfloat operations are bounded by 3 and hence each bigfloat operation incurs a rounding error of at most $3 \cdot 2^{-t}$. Thus

$$|e - z_0| \le 2/n_0! + 2n_0 \cdot 3 \cdot 2^{-t}.$$

[3] Additions, subtractions, and multiplications are exact if no rounding is performed and mantissas are allowed to have arbitrary length.

We want the right-hand side to be less than 2^{-m-1}; it will become clear in a short while why we want the error to be bounded by 2^{-m-1} and not just 2^{-m}. This can be achieved by making both terms less than 2^{-m-2}. For the first term this amounts to $2/n_0! \le 2^{-m-2}$. We choose n_0 minimal with this property and observe that if we use the expression $fac.length()| < m+3$ as the condition of our while loop then this n_0 will be the final value of n; $fac.length()$ returns the number of bits in the binary representation of fac. From $n_0! \ge 2^{n_0}$ and the fact that n_0 is minimal with $2/n_0! \le 2^{-m-2}$ we conclude $n_0 \le m+3$ and hence $6n_0 2^{-t} \le 6(m+3) \cdot 2^{-t} \le 2^{-m-2}$ if $t \ge 2m$; actually, $t \ge m + \log(m+3) + 5$ suffices. The following program implements this strategy and computes z_0 with $|e - z_0| \le 2^{-m-1}$.

We could output z_0, but z_0 is a number with $2m$ binary places and hence suggests a quality of approximation which we are not guaranteeing. Therefore, we round z_0 to the nearest number with a mantissa length of $m+3$ bits. Since $z_0 \le 3$ this will introduce an additional error of at most $3 \cdot 2^{-m-3} \le 2^{-m-1}$. We conclude that the program below computes the desired approximation of Euler's number.

```
bigfloat::set_precision(2*m);
bigfloat::set_rounding_mode(TOZERO);
bigfloat z = 2; integer fac = 2; int n = 2;
while ( fac.length() < m + 3 )
  {  // fac = n! and z approximates 1/0! + 1/1! + ... + 1/(n-1)!
    z = z + 1/bigfloat(fac);
    n++; fac = fac * n;
  }
// |z - e| <= 2^{m-1} at this point

z = round(z,m+3,TONEAREST);
}
```

Exercise 0.14: Show how to compute π with an error less than 2^{-200}. $\diamondsuit$

4.9 Notes

In notes we do historical notes, implementation notes, and pointers to additional material.

Error analysis for floating point computations was pioneered by Wilkinson [**?**]. Most books on numerical analysis contain a section on error analysis. Detailed discussions can be found in [**?**]. The analysis presented here is based on [12, 22, 23, 14].

The optimal choice of pivot in the orientation test is discussed in [13].

Error bounds similar to the ones derived in this lecture are used as floating point filters in the linear kernels of LEDA and CGAL. We discuss linear kernels in the next lecture.

Arbitrary precision integer and floating point arithmetic is provided by several software packages. Popular packages are the GNU Multiple Precision Arithmetic Library [15] and and the Java [19] classes BigInteger and BigDecimal. The former package is the most comprehensive.

The orientation test and the side-of-circle test amount to computing the sign of a determinant. In low dimensions, it is easy and efficient to expand the determinant into an arithmetic formula. In higher dimensions, this becomes infeasible. An obvious method for computing the sign of a determinant is to compute the value of the determinant and then take its sign. Better algorithms are discussed in [9, 1, 3].

The following sentence is from the LEDA book. We need a similar sentence in the introduction. Based on the bad experiences made by us and many others, we and others laid the theoretical foundations for correct and efficient implementations of geometric algorithms [20, 12, 11, 8, 29, 9, 22, 7, 6, 5, 4, 24, 10, 2, 30, 25, 3].

4.10 Material for the Lecture

It is not clear yet, where the following remarks should go.

Dynamic filters are more costly but also more precise than semi-dynamic filters. Observe that the computation of err_E in the case of an addition requires two additions and two multiplications. The computation of m_E requires only one addition. We concluded from our experiments in [22] that the additional cost is not warranted for the rational kernel.

We do use dynamic filters in the number type —real—, see Section **??**, since the cost of exact computation is very high for —reals— and hence a higher computation time for the filter is justified.

However, the necessary conditional branching could impair performance significantly. If one is willing to invest that time, one could also think of using an exact implementation scheme based on floating-point filter techniques, e.g. [12, 26], see [28] for results of an experimental comparison. Further details are beyond the scope of this paper.

Bibliography

[1] F. Avnaim, J.-D. Boissonnat, O. Devillers, and F. Preparata. Evaluating signs of determinants with floating point arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[2] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15(4):205–217, 1996. ISSN 0167-7055.

[3] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proceedings of 13th Annual ACM Symposium on Computational Geometry (SCG'97)*, pages 174–182, 1997.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 702–709, 1997. `www.mpi-sb.mpg.de/~mehlhorn/ftp/sepbound.ps`.

[5] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98)*, pages 175–183, 1998.

[6] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimenta l results. In *Proceedings of the 2nd Annual European Symposium on Algorithms - ESA'94*, volume 855 of Lecture Notes in Computer Science, pages 227–239. Springer, 1994.

[7] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 16–23, 1994.

[8] J. Canny, B. Donald, and G. Ressler. A rational rotation method for robust geometric algorithms. In A.-S. ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual ACM Symposium on Computational Geometry (SCG '92)*, pages 251–260, 1992.

[9] K. L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Sci.*, 33:387–395, 1992.

[10] O. Devillers, G. Liotta, F. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. Technical report, Center for Geometric Computing, Department of Computer Science, Brown Universi ty, 1997.

[11] S. Fortune. Robustness issues in geometric algorithms. In *Proceedings of the 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering (WACG'96)*, volume 1148 of Lecture Notes in Computer Science, pages 9–13, 1996.

[12] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15:223–248, 1996. preliminary version in ACM Conference on Computational Geometry 1993.

[13] S. J. Fortune. Numerical stability of algorithms for 2d Delaunay triangulations. *Int'l. J. Comput. Geometry and Appl.*, 5(1):193–213, 1995.

[14] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1997.

[15] GMP (GNU Multiple Precision Arithmetic Library). `http://gmplib.org/`.

[16] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1990.

[17] D. Goldberg. Corrigendum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 23(3):413–413, 1991.

[18] IEEE standard 754-1985 for binary floating-point arithmetic, 1987.

[19] Java. `http://www.java.com/en/`.

[20] M. Jünger, G. Reinelt, and D. Zepf. Computing correct Delaunay triangulations. *Computing*, 47:43–49, 1991.

[21] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7(7):595–596, 1963.

[22] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994. `www.mpi-sb.mpg.de/~mehlhorn/ftp/ifip94.ps`.

[23] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[24] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Computational Geometry*, 12(1-2):85–103, 1999.

[25] S. Schirra. Robustness and precision issues in geometric computation. to appear, preliminary version available as MPI report.

[26] J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[27] P. Sterbenz. *Floating Point Computation*. Prentice Hall, 1974.

[28] J. Tusch and S. Schirra. Experimental comparison of the cost of approximate and exact convex hull computation in the plane. In *CCCG*, 2006.

[29] C. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry (CCCG'93)*, pages 405–419, 1993.

[30] C. Yap and T. Dube. The exact computation paradigm. In *Computing in Euclidean Geometry II*. World Scientific Press, 1995.