Chapter 9.

Robustness

Writing robust programs for visual applications is an extremely challenging task. Programmers often perform low-level fine-tuning optimization on their codes, but tend to ignore robustness issues until their codes crash. Ideally, there should be no difference between a pseudocode algorithm and its fully-implemented C++ code. That is, we should certify our programs by formal mathematical proofs that those implementations perfectly code algorithms, without any flaw. Certifying programs in practice is extremely difficult, and tends to be limited for toy algorithms, such as computing the greatest common divisor of two integers. Even in those simple cases, we need computers to automatically output proofs, hoping thereby that the computer-generated "proof" program is not buggy itself! Thus, robustness should in fact be considered first when designing robust algorithms. In practice, robustness problems cannot be ignored when implementing algorithms manipulating geometric structures. This omnipresent robustness issue is illustrated in the research community by the fact that there has been a trend toward renaming "geometric computing" to the field of robust"computational geometry."

Section 9.1 presents various nonrobust scenarios and defines what robustness should be to handle those bad cases. We then briefly describe the IEEE 754 floating-point standard in Section 9.2. Section 9.3 gives an overview of filtering techniques that allows us to compute robustly using floating-point numbers. Section 9.4 introduces the framework of algebraic degrees of predicates, and explains why algorithms may be reconsidered from scratch in order to be robust. Finally, Section 9.5 gives an overview of main libraries and their interfaces for writing robust codes.

9.1 Identifying Code Weaknesses and Defining Robustness

Many problems occur in practice when implementing geometric algorithms. First we'll identify those problems and then define what an equivalent robust implementation should be:

Degeneracies. Often, for the sake of simplicity, we consider nondegenerate input. We design algorithms for the *general case*. That is, data is assumed to be in *general position*. There is no universal definition of "general position," as it depends on the problem the algorithm solves. For example, we assume that:

- no three points on the plane are collinear when computing the convex hull.
- no four points are cocircular when computing the Delaunay triangulation.
- no two points have the same x-abscissa when computing line segment intersections, etc.

However, in practice, we need to implement algorithms for all kinds of input, including all degenerate instances. Otherwise, imagine someone using your "general position" line segment intersection on axis-parallel segments of a VLSI circuitry! If we do not want to consider all possible instance cases, then we need to assert at least, that the input is in a general position, in order to avoid system crashes, infinite loops, or even worse, wrong output that we may believe true. The usual way to "theoretically" get rid of degenerate cases is to perturbate the input. Since randomly perturbated input may change the output too (think of a convex hull), perturbation methods need to be precisely controlled symbolically. That paradigm is called *simulation of simplicity*. Even wisely pertubating input does not solve all robustness problems. We'll further review the common implementation pitfalls.

Coherence. Assume we computed a convex hull of points on the plane. We expect the output to be a convex polygon. If not convex, assuming the algorithm is indeed correct, clearly something went wrong during the execution of the program (Figure 9.1). Thus, at the end of the program, it may be worth running a checking procedure, called *program checker*, that checks that the output is plausible. If the result is not correct, this obviously means that the structure has been miscomputed, and we can either recompute it robustly, or enforce the expected structure from the corrupted output. Thus, a program checker extends the traditional framework of testing programs. Furthermore, we do not need to provide a right suite of tests (a difficult task in its own) for validating a set of program outputs against correct results, computed using other robust means.

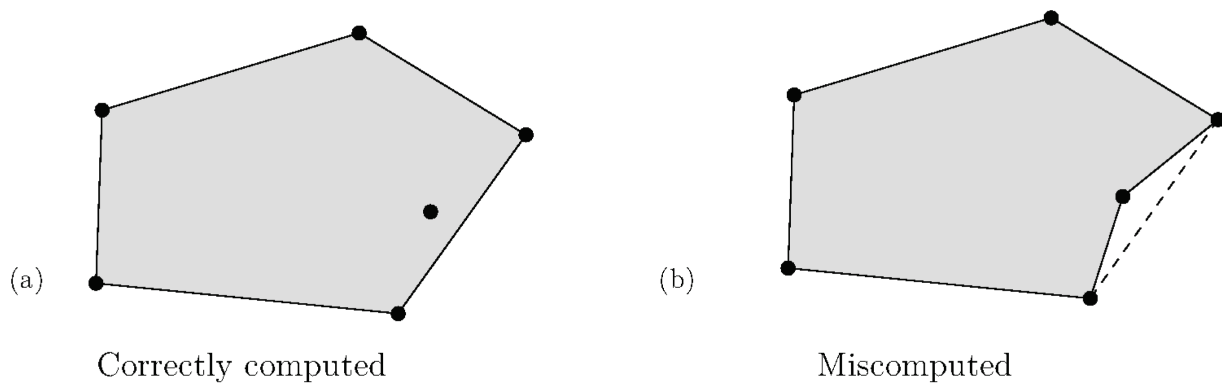

FIGURE 9.1 *Convex Hull: (a) a coherent structure. (b) a miscomputed nonconvex "convex hull."*

Since the checker is also a program, we face the egg and chicken problem: we need to design a checker for the checker, and so on. To break this infinite loop of checkers, we impose that program checkers are simpler than computing the solution. Moreover, we need the checker to be fast and undoubtedly correct. Let's consider Figure 9.1. A program checker for the convex hull problem may proceed as follows: (1) verify that the edge sequence is the facet graph of a piecewise linear curve, and (2) verify that this edge sequence is indeed the boundary of the convex hull of the point set (the right drawing of Figure 9.1 is not). Note that checking for local convexity of the edges of the output does not suffice to prove it is the correct result, as depicted in Figure 9.2. Let's give another coherence example. In computer vision, we know that theoretically the fundamental matrix is of rank two. Noisy data (pairs of matching points) could yield an estimated fundamental matrix of full rank (rank three). In that case, we rather find the closest rank-two matrix (see Section 3.5.7), as there might be no way to get a rank two matrix from that noisy input, even by considering exact computing. Instead of checking the output, the program could alternatively issue a *certificate* of correctness at the same time it delivers the output (see Section 6.1).

Numerical Stabilities. Programs differ essentially from algorithms because they use finite length sequences of bits to represent numbers. When designing algorithms, it is traditional to consider the *real-RAM model* of computation. The real-RAM model assumes that usual arithmetic operations on numbers are performed: (1) *exactly*, and (2) in *constant time.* More precisely, the real-RAM model of computation is described as follows:

- every memory location stores a real number
- access to any memory location is performed in constant time

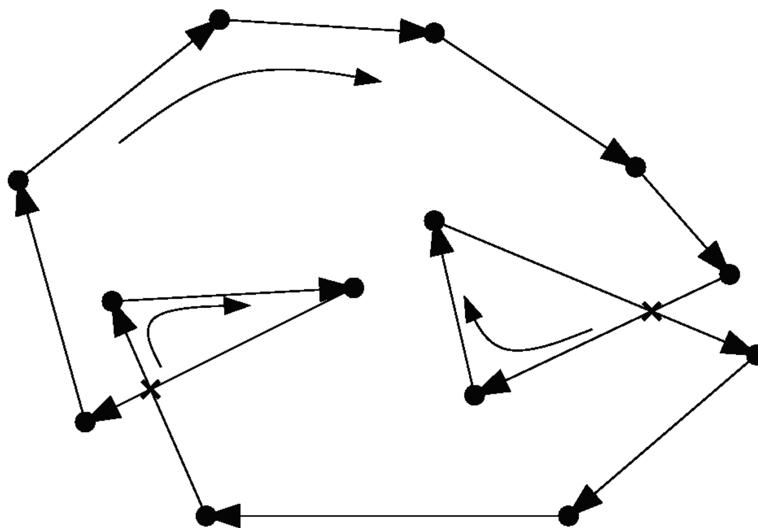

FIGURE 9.2 *Checking local convexity is not sufficient to assert the correctness of the convex hull output. A clockwise-oriented closed chain satisfies local convexity (CW orientation test) although it is not "globally" convex because of the self-intersections.*

- exact real-number arithmetic is computed in constant time:
  - comparisons: $<, \leq, =, \geq, >, \neq$
  - arithmetic operators: $+, -, \times, /$
  - algebraic and transcendental primitives: $\sqrt{\ }, \sqrt[k]{\ }, \sin, \cos, \exp, \log$.

Today, this real-RAM model is not realistic when implementing algorithms. Numerical inaccuracies are thus the fundamental issue to deal with for robustly implementing algorithms. For example, a real-RAM algorithm written in pseudocode can be shown to terminate whatever the input, but a straight C++ implementation may loop at infinity due to numerical errors. In particular, a common source of errors when implementing geometric algorithms is in the direct application of geometric theorems. For example, consider two lines of distinct slopes. These lines intersect exactly once in the continous Euclidean space. However, drawing those lines on the machine number grid,[1] we obtain *braided lines* that possibly intersect several times depending on the choice of the slope coefficients.

Once robustness is achieved, we then consider the low-level code optimizations, which improve running times.

[1] A process also known as line rasterization.

In the remainder, we assume that our input is precise, and not noisy. (In computer vision, noisy input is often modeled, say, by a Gaussian noise.) Our task is to execute an algorithm robustly, and not take into account noisy input. Consider a program implementing a correct algorithm: that is, an algorithm that formally returns the correct result for *any* input. If the corresponding program does not produce the correct result, this means that some error has occurred due to numerical inaccuracies. Let us analyze those sources of numerical errors. A geometric algorithm consists of *predicates* that control the flow of the algorithm (mainly in the `IF`, `SWITCH`, `REPEAT` and `WHILE` structures), and *constructions* that compute new geometric structures. At runtime, predicates are evaluated to decide which branch of the program to proceed on, while constructions create new geometric structures which may require us to calculate new numerical values prone to rounding errors (Section 9.3 further discusses those notions). For example, consider the predicate $\mathrm{Orient2D}(\mathbf{p},\mathbf{q},\mathbf{r})$ that returns +1, 0, or -1 depending on whether triangle $\triangle\mathbf{pqr}$ is oriented counterclockwise, clockwise, or is degenerated (i.e., the three points are exactly aligned). Namely, according to Section 8.3.1:

$$\mathrm{Orient2D}(\mathbf{p},\mathbf{q},\mathbf{r}) = \operatorname{sign}\det \begin{bmatrix} x_q - x_p & x_r - x_p \\ y_q - y_p & y_r - y_p \end{bmatrix} = \operatorname{sign}\det \mathbf{M_{pqr}}. \tag{9.1}$$

For points oriented clockwise (CW) the determinant is negative, for collinear points it is zero (ON), and it is positive for counterclockwise (CCW) orientation of the triplet:

$$\mathrm{Orient2D}(\mathbf{p},\mathbf{q},\mathbf{r}) = \begin{cases} \text{CCW} & \text{iff } \det \mathbf{M_{pqr}} > 0 \ , \\ \text{ON} & \text{iff } \det \mathbf{M_{pqr}} = 0 \ , \\ \text{CW} & \text{iff } \det \mathbf{M_{pqr}} < 0 \ . \end{cases} \tag{9.2}$$

Thus, when implementing geometric algorithms, for degenerate inputs (such as three collinear points or four cocircular points), programs have to correctly evaluate predicates that may return the zero value (ON). However, because of numerical inaccuracies (such as roundings, overflows, or underflows) when computing those predicate values, programs sometimes compute the wrong sign and therefore branch on the wrong instructions yielding incorrect construction results (potentially detected by coherence check) or system crash. Whenever possible, a program should thus avoid computing predicate signs by privileging topological deductions on the geometric combinatorial structures.

The traditional way to circumvent numerical inaccuracies is to use exact arithmetic. The idea of *exact arithmetic*, is that when adding two numbers a and b, we may need to increase the bit length of the result number by one bit (carry bit). Let $l(a)$ denote the bit length of number a. Then, if we want to compute $c = a + b$, we need to have $l(c) \leq \max\{l(a), l(b)\} + 1$ bits. Similarly, for multiplication $c = a \times b$, we bound the

required number of bits by adding the bit lengths of the inputs: $l(c) \leq l(a) + l(b)$. This framework defines the *exact integer arithmetic* that allows addition/subtraction and multiplication operations. Division is not supported by exact integer arithmetic. If divisions are used by the code, we need *exact rational arithmetic*. Exact rational arithmetic simply uses homogeneous representation of numbers to code for rational numbers (ratios of integers as described in Section 3.1.3). Thus, in rational arithmetic, a number f is stored as the ratio of a numerator n over a denominator d: $f = \frac{n}{d}$. Divisions and multiplications are now handled in the same way. Exact rational arithmetic is implemented using exact integer arithmetic, but requires us to double the bit length of numbers (because we handle a pair of integers). Since rational arithmetic costs time, we further speed up implementations using floating-point filters to avoid most of the time exact arithmetic. Floating-point numbers can either be manipulated in standard C++ single or double precision, or in multiprecision by using dedicated libraries such as CGAL (type `CGAL::MP_Float`), CORE (type `BigFloat`), GMP (type `mpf_t`), or LEDA (type `bigfloat`), described in Section 9.5. But before explaining the concept of floating-point filtering, let us review the floating-point standard.

9.2 IEEE 754 Binary Floating-point Standard

Writing integers in binary form is easy by using a string made of zeros and ones. An integer n is represented as a sequence of bits $\mathrm{bin}(b_{l-1}...b_2 b_1 b_0)$, such that:

$$n = \sum_{i=0}^{l-1} 2^i b_i. \tag{9.3}$$

Common *numeration systems* other than the binary system are: octal (base 8), decimal (base 10, our every day life numeration system), and hexadecimal (base 16). Computers internally crunch binary numbers.[2] For efficiency purposes, integer binary sequence lengths are handled in machine words. That is, the length of bit sequences is fixed: 16 bits (short ints), 32 bits (long ints), and more on graphics processor units (GPUs). The *big-endian* notation stores the most significant bytes of a number first, while the *little-endian* denotes the reverse byte order: $n = \mathrm{bin}(b_0 b_1 b_2 ... b_{l-1})$. Usually, we consider big-endian notation, although Intel® uses little-endian. Thus, both little-endian *and* big-endian options should be considered for code portability. Once the word length is chosen (say $l = 32$ bits), we encode negative integers using the frontmost bit, called *sign bit*. By conventions, we set the front bit to zero for positive integers, and to one for negative integers.

Thus, the range of integers (bounded by its maximum and minimum) we can represent with this fixed-size encoding of integers is $[-2^{l-1} + 1, 2^{l-1} - 1]$. Note that 0

[2] This is because of the on/off nature of transistors.

is coded twice, as +0 and −0. There is a better integer binary representation, called the *two's complement representation*, that allows to simplify the arithmetic circuits of signed integers. In complement notation, for each positive number n with leading bit set to zero, we define its negative number $(-n)$ such that $n + (-n) = 0$ is calculated using the 1-bit addition (a XOR[3] operation) without carry out. This is as if the leading bit of negative numbers has positional value -2^{l-1} (and not 2^{l-1}):

$$(-n) = -2^{l-1} + \sum_{i=0}^{l-2} 2^i b_i. \tag{9.4}$$

For example, using 3-bit integers, number 2 is encoded as 010, and its complement (−2) as 110. In 3-bit two's complement notation, we have the following numbers:

$$\begin{array}{lcl} 000 & \to & 0 \\ 001 & \to & 1 \\ 010 & \to & 2 \\ 011 & \to & 3 \\ 111 & \to & -1 \\ 110 & \to & -2 \\ 101 & \to & -3 \\ 100 & \to & -4 \end{array}$$

Thus, zero is represented only once, and the signed integer range is asymmetric: $[-2^{l-1}, 2^{l-1} - 1]$.

Expressing floating-point numbers is a little trickier. The main difficulty resides in choosing the right position for the decimal point. Thus, when designing a floating-point number format, we have to select a tradeoff for the range, precision, and complexity of arithmetic operations on those floating-point numbers. It was necessary to standardize a floating-point format, the IEEE 754 standard, so that scientific computations yield the same numerical results, independently of the programming language and platform used. The solution, pioneered by Kahan in the early 1980's, that IEEE has chosen is to use *scientific notation*. In scientific notation, numbers are written using both a *mantissa* (also called *significand*) and an *exponent* part. For example, 23121971 is written as 2.3121971×10^7. That is, any floating-point number f is written as $f = m \times 10^e$, where m is the mantissa and e denotes the exponent. Because computers crunch numbers in binary representations, floating-point numbers use powers of two instead of powers of ten. In summary, an effective encoding of a nonzero floating-point number f is as follows:

$$(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent}-\text{bias}}. \tag{9.5}$$

[3] a XOR b (eXclusive OR) is true if, and only if, exactly one of the two variables a or b is true.

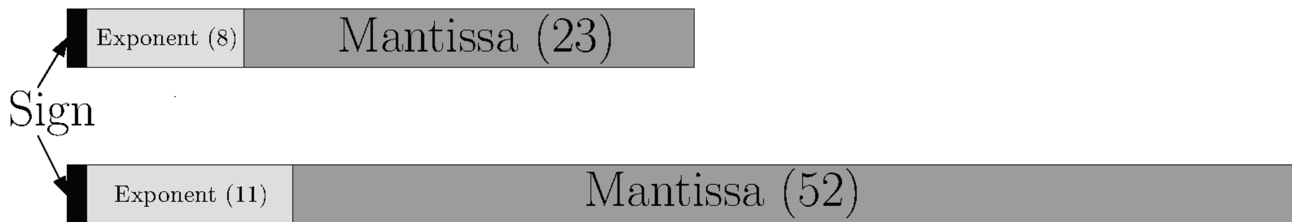

FIGURE 9.3 *Description of the encoding of the IEEE 754 single-precision (32 bits) and double-precision (64 bits) floating-point numbers.*

The sign of f is determined by the sign bit. For positive numbers, the sign bit is set to zero (we have $(-1)^0 = 1$). Conversely, for negative numbers, the sign bit is set to one (and $(-1)^1 = -1$). Moreover, we can always assume that the leading digit of the mantissa is one. Indeed, if not, we just have to increase the exponent. In that way, we can save an extra bit since it is enough to encode only for the remaining fractional part. That is, we consider the *normalized* floating-point number representation (see Eq. 9.5). Finally, the power of two is always set to an integer value (positive or negative), which we encode for efficiency as a positive integer that is added to some bias value. The bias allows to center around zero the exponent without requiring yet another extra sign bit (numerical order follows the lexicographic order). In summary, IEEE proposed two kinds of floating-point numbers representations in its IEEE 754 format, defined as follows (Figure 9.3):

Single-precision. Single-precision floats are encoded into a 32-bit string (bias value set to $127 = 2^7 - 1$): sign (1 bit), mantissa (23 bits), exponent (8 bits).

Double-precision. Double-precision floats are using twice as many bits (64 bits) with bias equal to $1023 = 2^{10} - 1$: sign (1 bit), mantissa (52 bits), exponent (11 bits).

For example, consider the decimal floating-point value 123456.7890123. Then, its IEEE 754 single-precision floating-point binary representation is:

$$\underbrace{0}_{+}\ \underbrace{10001111}_{143-127=16}\ \underbrace{(1.)11100010010000001100100}_{\text{Decimal value}=1.8838010},$$

and its IEEE 754 double-precision floating-point binary representation is:

$$\underbrace{0}_{+}\ \underbrace{10000001111}_{1039-1023=16}\ \underbrace{(1.)1110001001000000011001001111110010110101110010001010}_{\text{Decimal value}=1.8838011018722533}.$$

TABLE 9.1 *Single-precision floating-point number encodings: normalized, denormalized, and special numbers.*

Exponent	Mantissa	Encoded Number
1-254	Anything	Normalized floating-point number
0	Nonzero	Denormalized floating-point number
255	0	Infinity (inf)
255	Nonzero	Not-a-number (NaN)
0	0	0

Note that:
$1.8838011018722533 \times 2^{16} = 1.8838011018722533 \times 65563 \simeq 123456.7890123$ (with rounding).

The IEEE 754 standard also defines other 43-bit and 80-bit extended floating-point formats, but those are mostly used internally by microprocessors for rounding purposes. For example, in the IA-32 Intel architecture (commonly called x86), the processor arithmetic and logic unit (ALU) is based on the former 8087 floating-point coprocessor, and there is a limited set of floating-point registers that have internal formats 80 bits wide. IBM PowerPC® and MIPS Technologies MIPS® architectures have similar registers with dedicated floating-point instructions. Graphics card makers also provide floating-point formats on their GPUs: NVIDIA's GeForce® 6 series supports both 16-bit (1 sign bit, 10 mantissa bits, and 5 exponent bits) and 32-bit floating-point formats. ATI's Radeon® supports a 24-bit floating-point format (1 sign bit, 16 mantissa bits, and 7 exponent bits).

The IEEE 754 standard specifies *special* floating-point numbers too. For example, zero is a special floating-point number since it cannot be represented directly from the above explanation, because of the implicit leading bit in the mantissa (see Eq. 9.5). Thus, zero is rather represented by two zeros $+0$ and -0 (zero or one sign bit followed by 31 or 63 zeros).

Arithmetic on floating-point numbers can yield *underflow* or *overflow* situations. An underflow situation occurs whenever a number is so small that it cannot be represented using the finite length binary string. The converse situation is an overflow produced when adding two big numbers. In those cases, IEEE 754 chose to encode the inf symbol (infinity ∞) by setting all exponent bits to one, and all mantissa bits to zero. We summarize those special floating-point formats in Table 9.1.

A *denormalized floating-point number* has all its exponent bits to zero but the fractional part nonzero (otherwise, it would the interpreted as the special floating-point number zero). The value of a denormalized number is computed without assuming the implicit mantissa leading 1 bit. Thus, a denormalized floating-point

TABLE 9.2 *Smallest positive representable numbers and machine epsilons of the IEEE 754 format.*

	Single-precision	Double-precision
Normalized	$2^{-126}(\sim 1.175 \times 10^{-38})$	$2^{-1022}(\sim 2.225 \times 10^{-308})$
Denormalized	$2^{-127} \times 2^{-23} = 2^{-150}$	$2^{-1023} \times 2^{-52} = 2^{-1075}$
	$(\sim 7.0 \times 10^{-46})$	$(\sim 2.5 \times 10^{-324})$
Machine epsilon	$2^{-23}(\sim 1.19 \times 10^{-7})$	$2^{-52}(\sim 2.22 \times 10^{-16})$

number with sign bit s and fractional part f is assigned value $(-1)^s \times 0.f \times 2^{-126}$ for simple-precision, and $(-1)^s \times 0.f \times 2^{-1022}$ for double-precision. (We can interpret zero as a special extension of denormalized numbers.) For single-precision floating-point numbers, the smallest positive representable (denormalized) number is $2^{-127} \times 2^{-23} = 2^{-150}$ $(\sim 7.010^{-46})$. For double-precision, the smallest (denormalized) number is approximately 2.510^{-324} (2^{-1075}). Table 9.2 provides the smallest positive representable normalized and denormalized numbers for single-precision and double-precision floating-point numbers.

Even for special numbers, some arithmetic is still possible, such as:

$$\inf + \inf = \inf, \tag{9.6}$$

$$\frac{f}{\inf} = 0, \tag{9.7}$$

$$\frac{f}{0} = \inf. \tag{9.8}$$

But there are also cases when numbers cannot be defined, such as:

$$\frac{\inf}{\inf} = \boxed{?}, \tag{9.9}$$

$$0 \times \inf = \boxed{?}, \tag{9.10}$$

$$\sqrt{-\square} = \boxed{?}, \tag{9.11}$$

$$\square \bmod 0 = \boxed{?}, \tag{9.12}$$

$$\inf \bmod \square = \boxed{?}. \tag{9.13}$$

In these cases, numbers are classified using the label *Not a Number* (NaN). NaNs are represented by setting all the exponent bits to one, and the mantissa as any nonzero number (see Table 9.1).

Because the IEEE 754 floating-point standard only samples a finite number of the infinite set of reals $\mathbb{R}$, we need to round numbers. The standard defines four rounding options for snapping any real number to a close IEEE 754 floating-point number:

Round nearest. It is the usual rounding mode which chooses the nearest representable floating-point value (that has a low-order digit).

Truncation. Truncation simply discards the high-order digits to get the representable floating-point number (rounds toward zero).

Round up. Rounds toward plus infinity. That is, choose the closest larger floating-point number.

Round down. Rounds toward minus infinity. That is, choose the closest smaller floating-point number.

Rounding requires us to manage internally two extra bits, called the *guard bit* and *round bit*. These are stored on the right sides of the floating-point number representations. There is also the *carry bit* stored on the left. Using those extra bits, the IEEE 754 can provably do correct rounding. That is, the rounding rules for the IEEE 754 standard are guaranteed exact for $+$, $-$, $\times$, $/$, and $\sqrt{\ }$. This means that we compute as if we were using as many needed digits for exact results and then finally round.

To see the importance of the *guard* bit, we'll consider the following subtraction: $2 - 1.75 = 0.25$. Using 3-bit length notation for the mantissa part, we write:

$$1.00 \times 2^1 - 1.11 \times 2^0. \tag{9.14}$$

For performing subtraction, we first need to *align* the mantissa (and change accordingly the exponent part):

$$\begin{array}{cccccccc} + & 1 & . & 0 & 0 & \times & 2^1 \\ - & 0 & . & 1 & 1 & \times & 2^1 \\ \hline + & 0 & . & 0 & 1 & \times & 2^1 \end{array}$$

Then, we normalize 0.01×2^1 to 1.00×2^{-1} and find $\frac{1}{2}$, and not the correct result $\frac{1}{4}$. Using an extra guard bit, we avoid that problem, as shown below:

$$\begin{array}{ccccccccc} + & 1 & . & 0 & 0 & 0 & \times & 2^1 \\ - & 0 & . & 1 & 1 & 1 & \times & 2^1 \\ \hline + & 0 & . & 0 & 0 & 1 & \times & 2^1 \end{array}$$

That is, after normalization, $1.00 \times 2^{-2} = \frac{1}{4}$, the correct result.

The format also defines a *sticky bit* for rounding, to keep track whether there are any one bit beyond the guard and round bits during a sequence of floating-point operations.

In C++, there are two useful constants defined in the `<float.h>` header file: `FLT_EPSILON` (for single-precision) and `DBL_EPSILON` (for double-precision). Those constants are defined as the smallest single/double-precision floating-point number such that $1.0 + f \neq 1.0$. They are called *machine epsilons* (see Table 9.2): $2^{-23} \sim 1.19 \times 10^{-7}$ for single-precision, and $2^{-52} \sim 2.22 \times 10^{-16}$ for double-precision floating-point numbers. The smallest floating-point number is obtained in C++ as:

```
1 numeric_limits<float >::min()
```

Those constants are used as initialization values for filtering, described in the following section (see also Section 2.7.3).

Note that even if our floating-point numbers are well defined (that is not a NaN), the result can be "fairly" different from the real result. A typical example is to compute the area of a triangle using *Heron's formula*: given the side lengths a, b, and c of a triangle T, its area is mathematically calculated as:

$$\text{area}(T) = \sqrt{s(s-a)(s-b)(s-c)}, \tag{9.15}$$

where s is the half-perimeter: $s = \frac{a+b+c}{2}$ of triangle T.

Kahan showed the pitfalls of a naive implementation and proposed an accurate implementation using machine arithmetic. For example, let's take:

$$(a, b, c) = (31622.77662, 0.0155555, 31622.77661), \tag{9.16}$$

then the area is computed robustly as **245.95399999**480249 but a naive implementation using C++ `double` yields **245.953999**96791997 (same digits are emphasized in bold face). Even worse, consider $(a, b, c) = (10000, 5000.000001, 15000)$, then using the naive computation on `float` we get 0 but the exact truncated result is 612.37243572131004. Thus, if this triangle area computation is involved for determining the sign of a predicate, we may obtain two different algorithm workflows: the correct workflow (positive area), and the wrong workflow (degenerate triangle) that might yield disastrous consequences (looping program, system crash, etc.).

WWW	Additional source code or supplemental material is provided on the book's Web site: *www.charlesriver.com/Books/BookDetail.aspx?productID=117120* File `heron.cpp`

```
1 float a,b,c,heron,s;
2 double A,B,C, HERON, S;
3
4 a=10000; b=5000.000001; c=15000;
5 s=a+b+c;s/=2.0;
6 heron=s*(s-a)*(s-b)*(s-c);
7 heron=sqrt(heron);
8
9
10 A=10000; B=5000.000001; C=15000;
11 S=A+B+C;S/=2.0;
12 HERON=S*(S-A)*(S-B)*(S-C);
13 HERON=sqrt(HERON);
14
15 cout.precision(10);
16
17 cout<<"Heron's formula for computing the area of a triangle:\n";
18 cout <<"Area using single-precision floating-point numbers:";
19 cout <<heron<<endl;
20 // We find 0
21
22 cout <<"Area using double-precision floating-point numbers:";
23 cout <<HERON<<endl;
24 // We find 612.3725394
```

In summary, floating-point numbers are a subset of rationals unevenly distributed on the real line $\mathbb{R}$. They are not closed under basic operations because of the roundings, and do not satisfy the associativity nor distributivity properties. However, the IEEE 754 floating-point number standard correctly rounds usual operations and provides a format consistent across machines (including the handling of special numbers).

9.3 Filtering Predicates

A geometric algorithm consists of *predicates* that control the flow of the algorithm (mainly in the `IF`, `SWITCH`, `REPEAT`, and `WHILE` structures), and *constructions* that compute new geometric structures. At program runtime, predicates are evaluated to decide which branch to proceed on, while constructions create new structures, which may require us to calculate new numerical values prone to rounding errors if no exact arithmetic is used. That is, constructors build geometric embeddings of the combinatorial structures induced by the flow of the algorithm (say, a Delaunay triangulation or an arrangement of line segments). Typical combinatorial structures in computational geometry are the incidence graphs of faces of polytopes, line, or curve arrangement graphs (trapezoidal maps) or simplicial complexes. Thus, the output of any geometric algorithm is a combinatorial structure induced by predicates with an associated geometric embedding obtained from constructors.

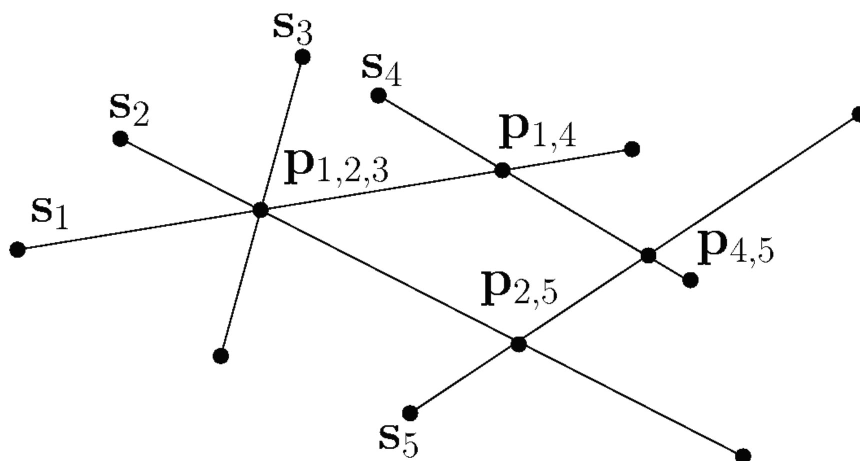

FIGURE 9.4 *Combinatorial structure defined by the intersection points of a set of line segments. The quad sequence* $\mathbf{p}_{1,2,3}$, $\mathbf{p}_{1,4}$, $\mathbf{p}_{4,5}$, $\mathbf{p}_{2,5}$ *is easily proven a closed simple polygon by looking at the index numbers of the intersection points. Any two consecutive points on the quad share a common line segment (see the respective point indices).*

Let us give some concrete examples. Computing the circumcenter and radius of the circle passing through three points is a geometric construction prone to numerical errors. To compute the convex hull of a planar point set, we do not need to compute new point coordinates, but rather select the extreme points as vertices of the convex hull. Then, the combinatorial structure of the hull is a convex polygon made of those selected extreme vertices linked by straight-line edges. Similarly, to find the intersection points of a set of segments, rather than explicitly creating new points resulting from the intersection of segments, we should rather store for each intersection points the (at least) two segments that yield this intersection point. The main advantage of this method is that we can now identify logical intersection points: two intersection points may be found distinct because of floating-point roundings although they represent the same logical intersection point. Also, this representation allows us to store intersection "segments" when some segments partially coincide in their interior.

Actually, it is known and mentioned in Section 2.8 that we can count the total number of intersection points faster than enumerating all of them. This means that there is a certain structure in the list of intersection points that can be compressed. Intersection points are not just plain unorganized intersection points, but rather critical locations of an *arrangement* of segments. For example, looking at Figure 9.4 we know that there is a closed convex quad in the arrangement of segments from

the combinatorial information stored at the intersection points. This combinatorial information manipulated on top of the point numerics is useful in various contexts. For example, if we need to compare two abscissa of intersection points, and we know that they share a common segment (by inspecting their indices), then the comparison test can be implemented more robustly (see also Section 9.4).

Being robust requires us to make the right decision at branching tests of programs. That is, the programs needs to calculate the correct signs $(+1, 0, -1)$ of mathematical expressions, in order to have the same control flow as the algorithm. We could compute those signs using exact arithmetic but this is time costly, and after all, we just want the sign and not the full value. So instead of using exact arithmetic, we calculate predicates using optimized operations on floating-point numbers. In most of the cases, we can guarantee the sign from floating-point computations. Thus, the idea is to calculate predicates as follows: First, we apply a fast floating-point computation. If the sign cannot be asserted correctly, then eventually we require exact arithmetic. Note that for the degenerated cases (sign 0), we often need exact arithmetic. This framework of evaluating predicate expressions is called *filtering*. It was shown experimentally on a 3D Delaunay triangulation implementation that the percentage of wrong sign answers of predicates `Orient3D` and `InSphere3D` (see Section 8.3.1) was zero for one million points randomly distributed inside the unit cube, but above 0.1% on a CAD model exhibiting many flat portions. Remember that the percentage measure is not a good clue of robustness since a single miscomputation can potentially yield a system crash, a looping program, or a wrong result. The following pseudocode exemplifies how filtering is used in the program control structures (sign of a predicate f):

```
RobustnessWithFiltering()
 1.  ◁ Start with standard floating-point computation ▷
 2.  precision = floating-point standard precision
 3.  ◁ Determine the sign of a predicate ▷
 4.  while precision < max_precision
 5.     do f = EvaluatePredicateExpression(precision)
 6.        if f > error_threshold(precision)
 7.           then return +1
 8.           else  if −f > error_threshold(precision)
 9.                    then return −1
10.                    else Increase precision
11.  ◁ At last step, use exact arithmetic ▷
12.  return EvaluatePredicateExpression(exact)
```

For example, considering the procedure RobustnessWithFiltering, we can initially start with single-precision floating-point numbers for first evaluating function $f(\cdot)$.

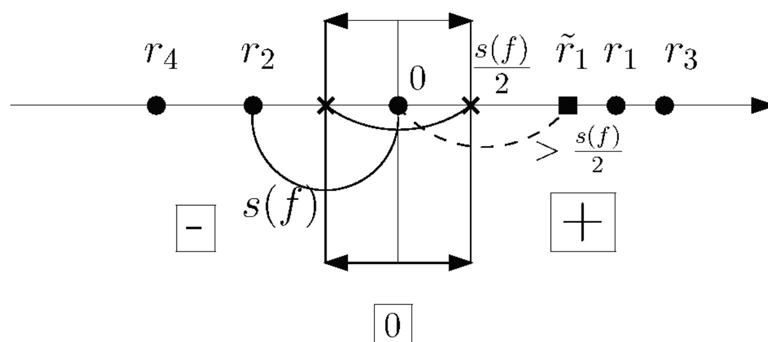

FIGURE 9.5 *Determining the sign of a rounded root value* $\tilde{r}_1$ *according to the separation bound* $s(f)$ *of the polynomial* f.

If the absolute value of the expression value is below an error threshold bound (depending on the current level of precision), then we need to increase precision to double-precision, extended multiprecision and finally arbitrary large precision (exact computation). This floating-point filtering method is quite effective in practice. Indeed, it was observed experimentally that a robust implementation with filtering slows down a nonrobust implementation by just 10%–40%. There are two drawbacks of filtering: first, filtering requires us to mathematically calculate the right error threshold values for the different precision levels, and second the source code for computing standard geometric predicates becomes clumsy. For example, Shewchuk uses a four-stage floating-point filtering technique that reuses computations of earlier stages for improving time performance. Shewchuk's predicate routines are complex. To give an idea, his `InSphere3D` has about 500 lines of code.[4]

We distinguish and further explain three types of predicate filters:

Static Filters. Static filtering is a simple technique that computes at *compile time* the error bounds. For predicates, we consider as zero any number below a constant threshold (usually chosen as a power of 2). This threshold depends on the separation bounds $s(f)$ of the polynomial roots r of f. To determine the sign of an albegraic expression, we first calculate the separation bound $s(f)$ of the polynomial f. Then, we evaluate a root $\tilde{r}$ with precision at least $\frac{s(f)}{2}$. Finally, we conclude that if $|\tilde{r}| > \frac{s(f)}{2}$ then $\mathrm{sign}(r) = \mathrm{sign}(\tilde{r})$, otherwise it is zero. Figure 9.5 depicts a case where $\mathrm{sign}(\tilde{r}_1) = \mathrm{sign}(r_1)$ because $|\tilde{r}_1| > \frac{s(f)}{2}$. The error bounds error() are based on the bit lengths of input data and intermediate numbers generated by a code. Let $u(f)$ denote an upper bound of the absolute value of f (say, used in a sign predicate). For example, the addition/subtraction and

[4]Check the routines at *http://www-2.cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c.*

multiplication rules on `double` type numbers is given by:

- $u(a \pm b) = u(a) + u(b)$,
- $u(a \times b) = u(a) \times u(b)$.

For the error bound, we have:

- $\text{error}(a \pm b) = \text{error}(a) + \text{error}(b) + u(a+b) \times 2^{-53}$,
- $\text{error}(a \times b) = \text{error}(a)u(b) + \text{error}(b)u(a) + u(a \times b) \times 2^{-53}$.

Static filters do not apply for divisions and become useless as $u(f)$ approaches the maximal representable floating-point number.

Semistatic Filters. In semistatic filtering, for each call of a predicate, we inspect the bit length representation of data at the time the predicate is called. Since we often call predicates on the difference of point coordinates that are in proximity, semistatic filtering can prove useful over static filtering. For example, when computing the Delaunay triangulation, an `InSphere3D` predicate is likely to have its argument points close to each other. So that the difference of them is better upper bounded compared to bound indicated by the static rule $u(a \pm b)$. Thus, semistatic filtering requires to add some piece of code to the program to compute on-the-fly error bounds.

Dynamic Filters. Dynamic filtering consists of estimating the error bounds for all predicate computations. The popular method is to use interval arithmetic, as it is done in CGAL (*http://www.cgal.org*). Interval arithmetic stores a pair of numbers for each computed predicate. Each pair defines a range where the true algebraic value[5] of the predicate is known to lie. Error bound intervals are updated whenever an arithmetic operation is carried on. In practice, interval arithmetic is quite easy to implement as a new number data type in C++. Thus, the use of dynamic filtering is quite transparent to programmers. To get a flavor of the mechanisms involved in interval arithmetic, we'll consider $[x]$ and $[y]$ two such "interval" numbers: $[x] = [\underline{x}, \overline{x}]$ and $[y] = [\underline{y}, \overline{y}]$. The interval error updating rules are as follows:

- $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
- $[x] - [y] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$
- $[x] \times [y] = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}]$

[5] An algebraic number is defined as a root solution of a polynomial with integer coefficients. For example, $\sqrt{2}$ is an algebraic number defined as the positive root solution to polynomial $x^2 = 2$.

- $\frac{[x]}{[y]} = (-\infty, \infty)$ if $0 \in [y]$, and $\frac{[x]}{[y]} = [x] \times [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}]$ otherwise
- $\sqrt{[x]} = [\sqrt{\underline{x}}, \sqrt{\overline{x}}]$ for $\underline{x} \geq 0$

Because $\underline{x}$ and $\overline{x}$ may not be contained in the finite set of floating-point numbers, those dynamic filter rules depend on the floating-point rounding modes. We consider intervals $[x] = [\underline{x}, \overline{x}]$, with $\underline{x}$ rounded toward minus infinity, and $\overline{x}$ rounded toward plus infinity.

Remember that in practice, filters are evaluated in cascading style for efficiency:

$$\text{static filter} \longrightarrow \text{semidynamic filter} \longrightarrow \text{dynamic filter.}$$

9.4 Predicate Degrees

We have stressed the crucial role of predicates for controlling a program flow, and presented the exact arithmetic paradigm for evaluating them. Moreover, we presented the floating-point filtering technique that avoids exactly computing the sign of mathematical expressions that could be decided without errors with fast standard floating-point operations. Here, we focus on comparing predicates with each other. Designing an algorithm can be done in an axiomatic fashion: First, we define the geometric predicates that define the control blocks (the axioms) and then build an algorithm to solve a specific task using them. Thus, the set of predicates we allow in our algorithm construction will directly affect the algorithm design and its implementation robustness. We review the predicate terminology pioneered by Liotta, Preparata, and Tamassia, in 1996.

- To each input data (point coordinate, etc.) we associate a variable.
- An *elementary predicate* has to compute the sign (-1, 0, +1) of a homogeneous multivariate polynomial defined on the input variables.
- To define the *degree d* of an elementary predicate, first we consider its irreducible factors of the predicate polynomial. Polynomials are factorized over the rationals. We remove all irreducible factors that have constant signs (say, $x^4 + 1 > 0$), and take the maximum degree of the remaining factors as the degree measure.
- A *predicate* is either elementary or expressed as a Boolean function of elementary predicates. The degree of a predicate is the maximum degree of its elementary predicates.

The *degree of an algorithm* is defined as the maximum degree of its predicates. The *degree of a problem* is then defined as the minimum degree of any algorithm's solving it.

Computational geometry has traditionally considered the real-RAM model of computation and ignored robustness issues of algorithm implementations. Today, researchers focus on the *geometric computing* paradigm that considers the bit-RAM model. That is, algorithms are designed and analyzed at the bit level. Thus, the complexity of an algorithm directly depends on the number of bits it requires (input and intermediate variables) to solve a problem. In that context, the degree d of an algorithm is a pertinent measure, as it is connected to the robustness/speed of implementations. Using this framework, we further define the *predicate arithmetic of degree d* which only allows us to compute the sign of the predicate polynomials of degree d, and the *exact arithmetic of degree d* model which computes exactly both the sign and the value of a predicate. For input on b bits, each monomial occurring in a predicate is upper bounded by $2^{(b+1)d}$. Exact arithmetic of degree d requires roughly db-bit exact arithmetic. Let m be the maximal number of variables occurring in a predicate and d the degree of an algorithm. Then this algorithm can be implemented using $p \leq d(b + 1 + \log m)$ bits in the exact arithmetic model of degree d. Note that there are many tricks for determining the sign of a polynomial quicker than evaluating its value. Some polynomials such as `OrientdD` or `InSpheredD` (see Section 8.3.1) can be expressed using determinants. Note that predicate `OrientdD` is of degree d. There have been tailored algorithms to compute only the sign of a determinant (see the Bibliographical Notes in Section 9.6). There are also some mathematical rules of thumb such as Descartes' rule of signs. Descartes' *rule of signs* is an elegant method for determining the maximum number of positive and negative real roots of a polynomial function $f(x)$. The maximum number of positive roots is found by counting the number p, the number of times the sign of monomial coefficients change when starting from the lowest to the highest monomial. Furthermore, the true number of positive roots can only be in $\{p, p-2, p-4, ..., 2, 0\}$. To find an upper bound on the number n of negative roots, we consider $f(-x)$ that monomial coefficients only invert sign for odd degrees, and apply the same counting. For example, consider $f(x) = 10x^6 + 6x^5 - 3x^4 + 2x^3 + x^2 - 100x$, then $p = 3$. Writing $f(-x) = 10x^6 - 6x^5 - 2x^3 + x^2 + 100x$, we get $n = 2$. Furthermore, real root isolations can be achieved using *Sturm sequences* that go beyond the scope of this book.

Orientation tests described in Chapter 8.3.1 are useful to build more complex predicates such as detecting whether a set of line segments intersect or not. For example, let us consider the following geometric problem to solve:

"Report all pairs of intersecting line segments."

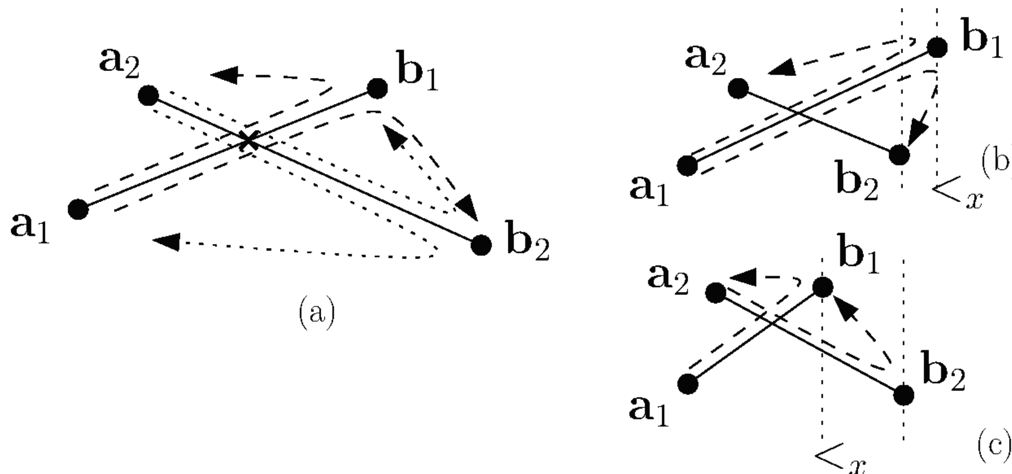

FIGURE 9.6 *Checking whether two line segments intersect or not.*

Let us first examine the basic primitive: decide whether two line segments intersect or not. Two line segments $[\mathbf{a}_1\mathbf{b}_1]$ and $[\mathbf{a}_2\mathbf{b}_2]$ intersect if and only if (see Figure 9.6):

$$\mathrm{Orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2) \neq \mathrm{Orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{b}_2)$$

and

$$\mathrm{Orient2D}(\mathbf{a}_2, \mathbf{b}_2, \mathbf{a}_1) \neq \mathrm{Orient2D}(\mathbf{a}_2, \mathbf{b}_2, \mathbf{b}_1).$$

The C++ code below shows an implementation of the basic line segment intersection predicate:

```
1 inline double drand(){return rand()/(double)RAND_MAX;}
2
3 class Point2D{
4 public: double x,y;
5
6     friend ostream &operator<<(ostream & o, const Point2D & p)
7     { o<<"("<<p.x<<","<<p.y<<")"; return o;}
8 };
9
10 // Orientation test: 2x2 determinant sign
11 #define ERR 1.0e-6
12 int Orient2D( const Point2D& p, const Point2D& q, const Point2D& r
     )
13   {
14     if ((q.x-p.x)*(r.y-p.y) > (r.x-p.x)*(q.y-p.y)+ERR) return CCW;
15     if ((q.x-p.x)*(r.y-p.y) < (r.x-p.x)*(q.y-p.y)-ERR) return CW;
16
17    return ON;
18   }
19
20 class Segment2D{
21 public: Point2D a,b;
```

```
22
23      Segment2D::RandomDraw()
24      {
25      a.x=drand();a.y=drand();
26      b.x=drand();b.y=drand();
27
28      // swap extremities
29      if (a.x>b.x) swap(a,b);
30      }
31
32      friend ostream &operator<<(ostream & o, const Segment2D & s)
33      {o<<"Segment [" <<s.a<<" ,"<<s.b<<" ]"; return o;}
34
35      bool Intersect(Segment2D s)
36      {
37      if (Orient2D(a,b,s.a)==Orient2D(a,b,s.b)) return false;
38      if (Orient2D(s.a,s.b,a)==Orient2D(s.a,s.b,b)) return false;
39      return true;
40      }
41  };
```

WWW	Additional source code or supplemental material is provided on the book's Web site: *www.charlesriver.com/Books/BookDetail.aspx?productID=117120* File: `segmentintersection-primitive.cpp`

This segment intersection test predicate can further be optimized so that it requires only two orientation tests. Assume without loss of generality (otherwise swap segments and/or endpoints using at most three comparisons) that:

$$\mathbf{a}_1 <_x \mathbf{b}_1, \qquad \mathbf{a}_2 <_x \mathbf{b}_2, \qquad \text{and} \qquad \mathbf{a}_1 <_x \mathbf{a}_2, \tag{9.17}$$

where $<_x$ denotes the x-coordinate increasing order. If $\mathbf{b}_1 <_x \mathbf{a}_2$ then clearly the two line segments do not intersect. Therefore, assume in the following that $\mathbf{a}_2 <_x \mathbf{b}_1$. Figure 9.6(b) and (c) depicts the two remaining combinatorial cases: $\mathbf{b}_2 <_x \mathbf{b}_1$ or $\mathbf{b}_1 <_x \mathbf{b}_2$.

If $\mathbf{b}_2 <_x \mathbf{b}_1$ then

$$[\mathbf{a}_1\mathbf{b}_1] \cap [\mathbf{a}_2\mathbf{b}_2] \neq \emptyset \Longleftrightarrow \text{Orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2) \neq \text{Orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{b}_2). \tag{9.18}$$

If $\mathbf{b}_1 <_x \mathbf{b}_2$ then

$$[\mathbf{a}_1\mathbf{b}_1] \cap [\mathbf{a}_2\mathbf{b}_2] \neq \emptyset \Longleftrightarrow \text{Orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2) = \text{Orient2D}(\mathbf{a}_2, \mathbf{b}_2, \mathbf{b}_1). \tag{9.19}$$

The C++ piece of code for this optimized predicate becomes:

```
1 // Assume a.x<s.a.x
2 // Only two orientation predicate evaluations
3     bool IntersectOptimize(Segment2D s)
4     {
5     // can be separated by a vertical line
6     if (b.x<s.a.x) return false;
7
8     if (s.b.x<b.x)
9     {
10    if (Orient2D(a,b,s.a)!=Orient2D(a,b,s.b)) return true;
11    else return false;
12    }
13    else
14    {
15    if (Orient2D(a,b,s.a)==Orient2D(s.a,s.b,b)) return true;
16    else return false;
17    }
18    }
```

Observe that the degrees of predicates quickly increase when predicates make use of intermediate geometric variables (say, for example, a predicate based on the vertices of the Voronoi diagram). Thus, we did not require to explicitly compute the intersection point to report its presence.

If necessary, the intersection point of two nonparallel straight lines:

$$L_1 : a_1x + b_1y + c_1 = 0 \qquad \text{and} \qquad L_2 : a_2x + b_2y + c_2 = 0, \tag{9.20}$$

is calculated using Cramer's determinant rule as:

$$(x_i, y_i) = \left(\frac{\det \begin{bmatrix} b_1 & c_1 \\ b_2 & c_2 \end{bmatrix}}{\det \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}}, \frac{\det \begin{bmatrix} a_1 & c_1 \\ a_2 & c_2 \end{bmatrix}}{\det \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}} \right). \tag{9.21}$$

Notice that $\det \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} = a_1b_2 - a_2b_1 = 0$ if, and only if, lines L_1 and L_2 are parallel, with slope $-\frac{a_1}{b_1} = -\frac{a_2}{b_2}$.

Note that in Section 3.1.3, we proved that two projective lines $\mathbf{l}_1 = \begin{bmatrix} a_1 & b_1 & c_1 \end{bmatrix}^T$ and $\mathbf{l}_2 = \begin{bmatrix} a_2 & b_2 & c_1 \end{bmatrix}^T$ intersect in projective point:

$$\mathbf{p} = \begin{bmatrix} b_1c_2 - b_2c_1 \\ a_2c_1 - a_1c_2 \\ a_1b_2 - a_2b_1 \end{bmatrix}. \tag{9.22}$$

This formula is compactly written using the cross product as: $\mathbf{p} = \mathbf{l}_1 \times \mathbf{l}_2$. If the two projective lines are parallel, the point of intersection is the ideal point associated with the line direction of slope $-\frac{a_1}{b_1} = -\frac{a_2}{b_2}$:

$$\mathbf{p} = \begin{bmatrix} b_1c_2 - b_2c_1 \\ a_2c_1 - a_1c_2 \\ 0 \end{bmatrix}. \tag{9.23}$$

Yet there is a better approach to retrieve the intersection point, if it exists: consider the primitive orient2D($\mathbf{p}, \mathbf{q}, \mathbf{r}$) (small "o") that reports the value of the following determinant:

$$\text{orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \det \begin{bmatrix} x_q - x_p & x_r - x_p \\ y_q - y_p & y_r - y_p \end{bmatrix} = \det \mathbf{M}_{\mathbf{pqr}}. \tag{9.24}$$

Thus, we have:

$$\text{Orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \text{sign orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}). \tag{9.25}$$

Then, the intersection point $\mathbf{p}$ is:

$$\mathbf{p} = \mathbf{a}_1 + \lambda(\mathbf{b}_1 - \mathbf{a}_1), \tag{9.26}$$

with:

$$\lambda = \frac{\text{orient2D}(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_2)}{\text{orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{b}_2) - \text{orient2D}(\mathbf{a}_1, \mathbf{b}_1, \mathbf{a}_2)}. \tag{9.27}$$

Now, let us go back to our original problem: "Report all pairs of intersecting line segments." The seminal sweep-line intersection algorithm of Bentley and Ottman described in Chapter 2.4.2 reports the intersection of n line segments in output-sensitive $O((n+k)\log n)$ time. Implementing this algorithm using floating-point arithmetic yields very unstable behavior. A careful analysis of its predicates reveals that it is of degree five. Using single-precision floating-point numbers (23 bits in the mantissa), this means that we can guarantee only the implementation for input segments encoded on $\lfloor \frac{23}{5} \rfloor = 4$ bits. (there are only $2^4 = 16$ distinct coordinates then). Of course, in practice the bad configurations rarely happen but this doesn't mean that they don't exist. Now, let us look at the naive quadratic algorithm that consists in checking pairwise segments. This algorithm is of degree two, much better than five. In computational geometry, there has been renewed interest in revisiting traditional problems (Voronoi/Delaunay structures, ray/polygon intersections, nearest neighbor queries, etc.) with this new complexity degree indicator. To conclude the line segment intersection problem, let us mention that the current best solution is the algorithm provided by Balaban in 1995. Balaban's algorithm runs in $O(n \log n + k)$ time using

degree three. There is no known algorithm of degree two matching this bound (but a quadratic time degree-two naive algorithm exists).

9.5 Overview of Libraries

We have seen that implementing robust geometric programs requires careful handling of number data types, error bounds, and arithmetic operations. Fortunately, there are a lot of reusable pieces of code. These have been conveniently organized into libraries. Using finely optimized libraries allows us to ignore the nitty-gritty details, while still keeping control on the tradeoff between robustness and time efficiency. The ideal library usage scenario is illustrated in Figure 9.7: a robust-compliant compiler takes as input a traditional C++ source code (with eventual annotations) and a preference file that describes the kind of robustness we look for in the compiled code. Then, this meta compiler preprocesses the C++ source code to modify variable types and predicate evaluation procedures (filtering) to generate another C++ code. This last code is finally compiled using our standard C++ compiler, and linked with tailored libraries for robustness. Ultimately, robustness should be directly programmable using language instructions.

We present the most common libraries:

GMP. The GNU Multiple Precision Arithmetic Library (GMP) is the standard implementation of arbitrary length number types. GMP is widely used and acclaimed, and has gained a lot from fine-tuned optimization over the years. GMP (version 4.1.4) can be downloaded from the Internet at *http://www.swox.com/gmp/*.

LEDA. Historically, LEDA pioneered the software engineering aspect of handling numerical inaccuracies in geometric algorithms. LEDA provides arbitrary length integers `leda::integer`, rational numbers `leda::rational`, and algebraic real numbers `leda::real`. The `real` number data type provides mathematical correct results on expressions including square roots (and k-th roots $\sqrt[k]{\ }$). LEDA also supports big floating-point numbers, floating-point filtering, and interval arithmetic. The `bigfloat` package of LEDA calculates efficiently algebraic numbers using an adaptive evaluation of an expression directed acyclic graph (DAG) that reuses as much as possible arithmetic portions of formulas. LEDA (version 5.0) can be downloaded from *http://www.algorithmic-solutions.com/*.

CGAL. CGAL includes a geometric kernel with dynamic filtering implemented using interval arithmetic. In case the sign of an expression $[e]$ cannot be answered (that is, $0 \in [e]$), a C++ exception is thrown, and the calculation is done using exact arithmetic. CGAL provides a C++ wrapper for using GMP. CGAL

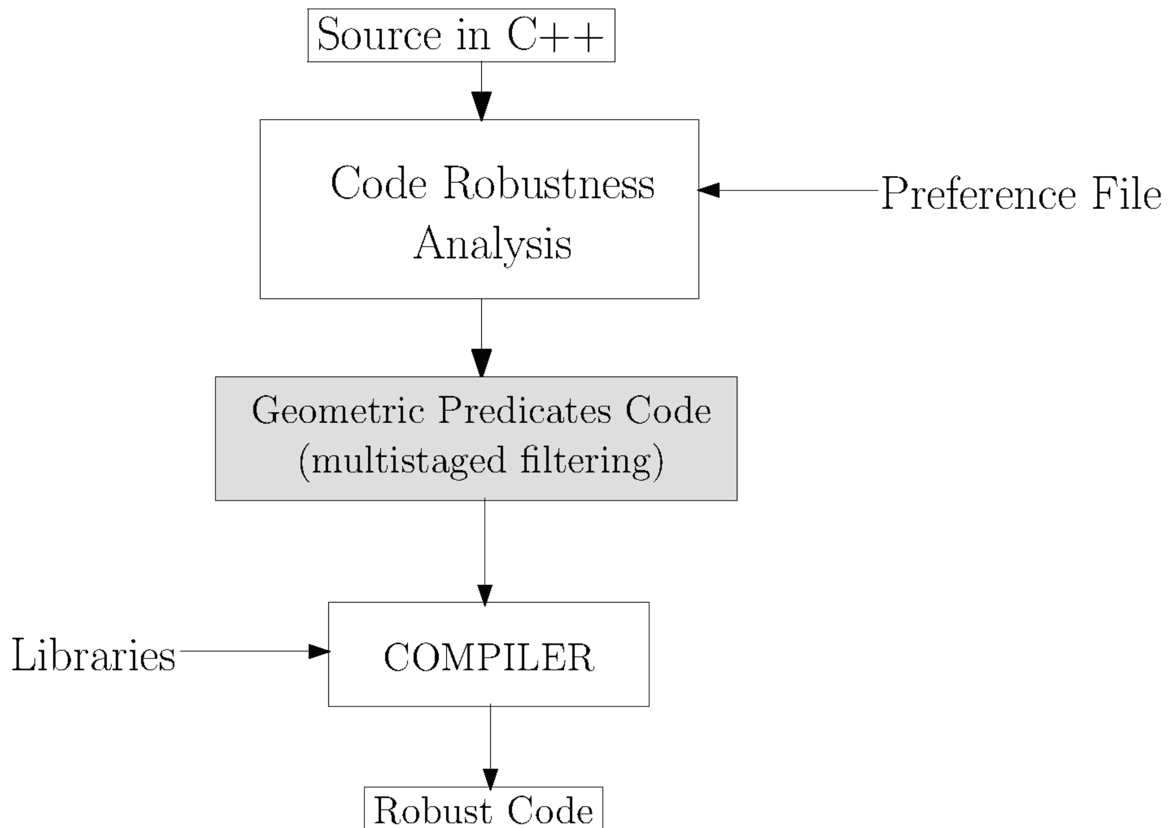

FIGURE 9.7 *Libraries often provide compiler preprocessors to analyze the code variables and predicates, and generate robust geometric predicates according to preference settings.*

also proposes semistatic filters for usual geometric predicates. Those filters are detailed in the online file *Class_Filtered_Kernel.html*. CGAL (version 3.1) is available at *http://cgal.org/*.

CORE. The key idea of the CORE library is to let us choose the tolerance level of our code. Level 1 just uses standard `double` for variables. Level 2 improves over level 1 by assigning once for all the length representation of numbers. Level 3 guarantees veracity of the first k bits of variables. Note that for correct predicate signs, we require level 3 with 1 bit certified accuracy. Finally, level 4 offers a mix of possibilities based on the previous levels. The CORE level is easily set in programs, using the following syntax (here, we set CORE robustness to level three):

```
#define CORE_LEVEL 3
#include "CORE/CORE.h"
```

Moreover, CORE can also be used in CGAL. CORE (version 1.7) can be

downloaded at *http://www.cs.nyu.edu/exact/core/*.

To conclude, robustness of programs should be tackled depending on the programmer's profile:

Novice to Intermediate Level. Programmers consider implementing real-RAM algorithms as they are described in textbooks or other materials, and handle robustness issues by using specific libraries when programming algorithms. This programmation paradigm requires us preferably to have some basic knowledge of the underlying arithmetic kernels of libraries. Hopefully, in the future, we will have programming languages that contain special standardized instructions for setting the level of robustness. With such language compilers, programs will be closer to algorithms than they are today.

Algorithmician Level. Algorithmicians should consider robustness issues from the very beginning task of designing algorithms using the bit or machine word model complexity. This framework allows us to define the basic set of predicates that define the control flow of algorithms. The main disadvantage of this approach is that we need to care about the nitty-gritty arithmetic details.

To give an analogy with programming recursive procedures, let's consider computing Fibonacci sequences (see Section 2.1.2). We can either write the program using exponential recursion or linear recursion. This corresponds to the "Novice to Intermediate Level" profile, where the compiler handles the low-level code (that is, managing the stack). However, we can also redesign the algorithm using a simple iterative approach based on the mathematical fact that:

$$F(2n-1) = F(n)^2 + F(n-1)^2. \tag{9.28}$$

This means that whatever the parity of n, $F(n)$ and $F(n-1)$ can be found in one step from $F(\lfloor \frac{n}{2} \rfloor)$ and $F(\lfloor \frac{n}{2} \rfloor - 1)$. This new algorithm has complexity $O(\log n)$ for computing the nth Fibonacci number. This latter scheme corresponds to the "Algorithmician Level."

9.6 Bibliographical Notes

The IEEE 754-1985 ISO/IEC 559 standard for binary floating-point arithmetic is implemented in most CPU arithmetic and logic units (ALUs). Because of the rounding rules, computing with IEEE 754 is more time consuming than using "quick" floating-point numbers. For example, we need to turn on the IEEE 754 option in `g++` using the flag `-mieee`. To test whether your code (and CPU[6]) is compliant

[6] In 1995, Intel had a massive recall of its Pentium® chip due to a floating-point divide flaw in the chip design.

or not, use the IEEE 754 floating-point test software of Beebe, available online at *http://www.math.utah.edu/~beebe/software/ieee/* (see also *http://www.win.ua.ac.be/~cant/ieeecc754.html*). The floating-point standard is briefly reviewed in [135], and described at length in the manual [259]. Interested readers can get the current activity status of the IEEE working group online, at *http://grouper.ieee.org/groups/754/*. Trigonometric functions are generally computed in hardware using CORDIC algorithms. CORDIC stands for COordinate Rotation DIgital Computer. The seminal CORDIC algorithm was proposed by Volter in 1959 [334], as an iterative method for performing arbitrary angle vector rotations using only shifts and adds. Further references to CORDIC algorithms are found by visiting online the CORDIC FAQ at *http://www.dspguru.com/info/faqs/cordic.htm*. See also the IEEE 854 standard that is radix-independent (i.e., allow any base).

In computer graphics, it is known that the nonlinearity of the perspective division eventually yields inaccurate results in the integer-based z-buffer. This phenomenon occurs depending on whether the setting of the minimum and maximum z range was done properly or not (see the definition of the perspective frustrum in Chapter 3.3.4). An alternative to the z-buffer is the w-buffer [35, 192], which is floating-point based.

Implementing robust computational geometry algorithms is known to be a difficult task, as attested by the early works by Sugihara and Iri [321]. They concentrated on flawlessly computing the Voronoi diagrams of one million points. Guibas et al. [286] considered imprecise input in their *epsilon geometry* paper. Edeslbrunner and Mücke described their simulation of simplicity paradigm [107] using symbolic perturbation. Melhorn et al. [225] described the framework of geometric program checkers, to detect whether a geometric program was executed correctly or not. The idea of program checkers is due to Blum and Kanna [38], in 1989.

Shewchuk presented its efficient adaptive precision floating-point filtering technique in [303]. (Some usual 2D/3D geometric predicates using his technique are available online, at *http://www.cs.berkeley.edu/~jrs/*.) Fortune and Van Wyk [123] designed the `LN` language for generating automatically geometric filters common to many predicates. Burninkel et al. [63] developed another expression compiler called `EXPCOMP` for automatically handling numerical inaccuracies when implementing geometric algorithms. Their use of a semistatic filter provides the speed of usual static filters, but is yet highly customizable (as dynamic filters are).

Goodrich et al. [139] reported an algorithm for efficiently snap rounding 2D/3D line segments. Brönnimann et al. [56] described the design of efficient dynamic filters using interval arithmetic in computational geometry. Devillers and Pion [98] focused on robustly computing the Delaunay triangulation. Brönnimann et al. [59] proposed a method that determines the sign of a multivariate polynomial with integer coefficients using modular arithmetic. Karamcheti et al. [181] discuss their CORE library in their paper. The book by Mehlhorn and Nahër on LEDA [224] is a good reference for

programming geometric algorithms using different arithmetics. The paper by Kettner et al. [182] gives a good overview of the common pitfalls when robustly implementing algorithms in computational geometry. They provide detailed analysis of "why" and "when" programs crash.

In computational geometry, predicates are often described using determinants (such as predicates `InSpheredD`, `OrientdD`, etc.) We do not need to compute the exact determinant but rather its sign. Computing exactly a $d \times d$ determinant with integer input encoded on b bits require $db + d \log d$ bits. Clarkson [81] presented an algorithm that robustly evaluates the sign of a determinant of a $d \times d$ matrix using $1.5d + 2b$ bits, where b is the bit complexity of the input. Avnaim et al. [18] presented a geometric algorithm to compute the sign of determinants using single-precision integer arithmetic. The number of necessary bits can be reduced to $d + b + \log d$ bits if only the sign is considered.

Boissonnat and Preparata [45] described a robust plane sweep algorithm (similar in spirit to the Bentley-Ottman's algorithm [31]) for reporting the intersections of a set of line segments, using the notion of algebraic degree of predicates. That notion of algebraic degrees of predicates in computational geometry algorithms was first introduced by Liotta et al. [205].

Computing Fibonacci numbers in logarithmic time was presented by Gries and Levin [144] in 1980.