Contents

4

Numbers and Matrices

Numbers are at the origin of computing. We all learn about integers, rationals, and real numbers during our education. Unfortunately, the number types *int*, *float*, and *double* provided by C++ are only crude approximations of their mathematical counterparts: there are only finitely many numbers of each type and for floats and doubles the arithmetic incurs rounding errors. LEDA offers the additional number types *integer*, *rational*, *bigfloat*, and *real*. The first two are the exact realization of the corresponding mathematical types and the latter two are better approximations of the real numbers. Vectors and matrices are one- and two-dimensional arrays of numbers, respectively. They provide the basic operations of linear algebra.

4.1 Integers

C++ provides the integral types *short*, *int*, and *long*. All three types come 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 $w = 64$. Unsigned ints and signed ints use w bits, shorts use at most that many bits, and longs use at least that many bits.

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`], where `MININT` and `MAXINT` are predefined constants; under UNIX they are available in the systems file `limits.h`. On most machines signed integers are represented in two's complement. Then $\texttt{MININT} = -2^{w-1}$ and $\texttt{MAXINT} = 2^{w-1} - 1$. The conversion from signed ints to unsigned ints adds a suitable multiple of m so as to bring the number into the interval $[0 \,..\, m-1]$. If numbers are represented in two's complement this conversion does not change the bit pattern. The conversion from unsigned int to signed int is machine dependent.

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 implementation dependent, in particular, it is not guaranteed that they lead to a runtime error, in fact they usually do not. On the author's workstations the summation `MAXINT + MAXINT` has result -2, since adding $011\ldots1$ to itself yields $11\ldots10$, which is the representation of -2 in two's complement. We give an example of the disastrous effect that an undetected overflow might have.

Some network algorithms are easier to state if the integers are augmented by the value ∞. For example, in a shortest path algorithm it is convenient to initialize the distance labels to ∞. In an implementation it is tempting to use `MAXINT` as the implementation of ∞ and to forget that it does not quite have the properties of ∞. In particular, `MAXINT + 1 = MININT` on the author's workstations which is drastically different from mathematics' $\infty+1=\infty$. This difference led to the following error in one of the first author's programs[1]. He implemented Dijkstra's shortest path algorithm (its working is discussed in Section 6.6) as follows:

```
void DIJKSTRA(const graph& G, node s, const edge_array<int>& cost,
              node_array<int>& dist)
{ node_pq PQ(G);
  node v; edge e;
  forall_nodes(v,G) dist[v] = MAXINT;
  dist[s] = 0;
  forall_nodes(v,G) PQ.insert(v,dist[v]);
  while (!PQ.empty())
  { node v = PQ.delete_min();
    forall_adj_edges(e,v)
    { node w = G.target(e);
      if (dist[v] + cost[e] < dist[w])
      { dist[w] = dist[v] + cost[e];
        PQ.decrease_p(w,dist[w]);
      }
    }
  }
}
```

This program works fine when all nodes are reachable from s and all edge costs are in $[0\,..\,\texttt{MAXINT}/n]$, where n is the number of nodes of G. However, consider the execution on the graph shown in Figure 4.1. When node v is removed from the queue, we have $dist[v] = dist[w] = \texttt{MAXINT}$. We compute `MAXINT + 1` which is `MININT` and hence decrease w's distance to `MININT`, a serious error. A correct implementation inserts only s into the queue initially and replaces the innermost block by

[1] The second author insists that he has never made this particular mistake.

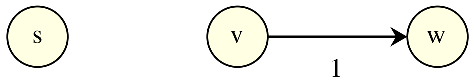

Figure 4.1 An example, where the naive use of `MAXINT` as a substitute for ∞ in Dijkstra's algorithm has a disastrous effect.

```
int c = dist[v] + cost[e];
if (dist[w] == MAXINT) PQ.insert(w,c);
else PQ.decrease_p(w,c);
dist[w] = c;
```

We come to the LEDA type *integer*. It realizes the mathematical type integer. The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$ (unary), $++$, $--$, the modulus operation (%, %=), bitwise AND (&&, &&=), bitwise OR (||, ||=), the complement operator ($\sim$), the shift operators ($\ll$, $\gg$), the comparison operators $<$, $\leq$, $>$, $\geq$, $==$, $!=$, and the stream operators are available. These operations never overflow and always yield the exact result. Of course, they may run out of memory. The following program computes the product of the first n integers.

```
integer factorial(int n) // computes 1 * 2 * ... * n
{ integer fac = 1; //automatic conversion from int
  for (int i = 2; i <= n; i++)  fac = fac*i;
  return fac;
}
```

Integers also provide some useful mathematical functions, e.g., $sqrt(a)$ returns $\lfloor\sqrt{a}\rfloor$, $log(a)$ returns $\lfloor\log a\rfloor$, and $gcd(a, b)$ returns the greatest common divisor of a and b. We refer the reader to the manual pages for a complete listing.

Integers are essentially implemented by a vector of unsigned longs. The sign and the size are stored in extra variables. The implementation of integers is very efficient and compares well with other implementations. This is particularly true on SPARC machines since we have implemented several time critical functions not only in C++ but also in SPARC assembler code. When integers are used on SPARC machines the faster assembler code is executed. The running time of addition is linear and the running time of multiplication is $O(L^{\log 3})$, where L is the length of the operands. The following program verifies the latter fact experimentally. It repeatedly squares an integer n and measures the time needed for each squaring operation. In each iteration it prints the current length of n (= number of binary digits), the time needed for the iteration and the quotient of the running time of this and the previous iteration.

⟨*multiplication_times*⟩≡

```
main()
{ integer n;
  ⟨multiplication times: read n⟩
  int i;
```

n	Running time	T/T_prev
167587	0.63	3
335173	1.88	2.984
670346	5.74	3.053
1340691	17.43	3.037

Table 4.1 The time required to multiply two n bit integers. The multiplication times demo allows you to perform your own experiments.

```
  for (i = 0; i < 11; i++) n = n * n;
  float T_prev = 0;
  for (i = 0; i <= 5; i++)
  { float T = used_time();
    n = n * n;
    T = used_time(T);
    ⟨multiplication times: report times⟩
    T_prev = T;
  }
}
```

Table 4.1 shows a sample output of this program. Since n is squared in each iteration, its length L essentially doubles in each iteration. Thus, if the running time of an iteration is $c \cdot L^{\alpha}$ for some constants c and α then the running time of the next iteration is $c \cdot (2L)^{\alpha}$ and hence the quotient is $c \cdot (2L)^{\alpha}/(c \cdot L^{\alpha}) = 2^{\alpha}$. The measured quotient is about 3. Thus, $\alpha \approx \log 3$.

Integers are used a lot in LEDA's geometric algorithms. We briefly hint at the use now and treat it in detail in Chapter 8. Consider three points p, q, and r in the plane and let l denote the line through p and q and oriented from p to q. For any point s use s_1 and s_2 to denote its Euclidean coordinates. The test of whether r lies to the right of l, on l, or to the left of l is tantamount to determining the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 \\ p_1 & q_1 & r_1 \\ p_2 & q_2 & r_2 \end{vmatrix}.$$

If the coordinates of our points are floating point numbers and the determinant[2] is evaluated

[2] Note that the determinant is zero if and only if the third column is a linear combination of the first two columns, i.e., if there are reals λ and μ such that $\lambda + \mu = 1$ and $r = \lambda p + \mu q$. In other words, if $r = p + \mu(q - p)$ for some μ. This shows that the determinant is zero if and only if r lies on the line through p and q. We still need to argue that the sign distinguishes the two half-planes defined by the line. Consider two points r and r' and the line segment from r to r'. The value of the determinant changes continuously as one moves from r to r' and hence assumes value 0 if r and r' lie on different sides of the line and does not assume value 0 if r and r' lie on the same side of the line. Since the determinant is a linear function we conclude that the two sides of the line are distinguished by the sign of the determinant.

with floating point arithmetic we may incur rounding error and determine the sign of the determinant incorrectly. This is a frequent source of error in the implementation of geometric algorithms, as we will see in Sections 4.4 and 8.6 . If the coordinates are integers then the determinant can be evaluated exactly and the correct sign can be determined. This feature facilitates the correct implementation of geometric algorithms enormously.

Exercises for 4.1

1 Write a procedure *random_integer*(*int* L) that returns a random integer of length L.

2 The greatest common divisor of two numbers x and y with $x \geq y \geq 0$ can be computed by the recursion $\gcd(x, y) = x$ if $y = 0$ and $\gcd(x, y) = \gcd(y, x \bmod y)$ if $y > 0$. Implement this algorithm, run it on integers of various lengths, and count the number of recursive calls. Relate the number of recursive calls to the length of y. 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 $x_i = x_{i-2} \bmod x_{i-1}$. 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 $x_{i-2} \geq x_{i-1} + x_i$ for $i < k$. Conclude that x_{k-j} is at least as large as the j-th Fibonacci number.

3 The standard algorithm for multiplying two L-bit integers has running time $O(L^2)$. LEDA uses the so-called Karatsuba-method ([Kar63]) that 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. Implement Karatsuba's algorithm and time it as described in the text. Compare to the member function *operator*$*$.

4 In program ⟨*multiplication_times.c*⟩ let *n_old* be the value of n before the assignment $n = n * n$. Extend the program such that it also computes $n/$*n_old* and *sqrt*(n) in each iteration. Measure the execution times and compute quotients of successive execution times. Try to explain your findings.

5 Develop algorithms for integer division and integer square root based on Newton's iteration.

4.2 Rational Numbers

A rational number is the quotient of two integers. Well, that is the mathematical definition and it is also the definition in LEDA. The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$ (unary), $++$, $--$ are available on rationals. In addition, there are functions to extract the numerator and denominator, to cancel out the greatest common divisor of numerator and denominator, to compute squares and powers, to round rationals to integers, and many others.

LEDA's rational numbers are not necessarily normalized, i.e., numerator and denominator of a rational number may have a common factor. A call *p.normalize*( ) normalizes p.

This involves a gcd-computation to find the common factor in numerator and denominator and two divisions to remove them. Since normalization is a fairly costly process we do not do it automatically. It is, however, advisable to do it once in a while in a computation involving rational numbers.

Exercises for 4.2

1 Write a program to solve linear systems of equations using Gaussian elimination. Use rational numbers as the underlying number type. Make two versions of the program: in one version you keep all intermediate results in reduced form by calling *x.normalize*( ) for each intermediate result and in the other version you make no attempt to keep the numbers normalized. Run examples and determine the lengths of the numerators and denominators in the solution vector.

2 Investigate the question raised in the first item theoretically. Assume that all coefficients of the linear system are integers of length at most L and let n be the number of equations in the system. Show that the entries of the solution vector can be expressed as rational numbers in which the lengths of the numerator and the denominator are bounded by a polynomial in n and L. (Hint: Show first that the value of an n by n determinant of a matrix with integer entries of absolute value at most 2^L is bounded by $n!2^{nL}$. Then use Cramer's rule to express the entries of the solution vector as quotients of determinants.) Extend the result to all intermediate results occurring in Gaussian elimination. Conclude that Gaussian elimination has running time polynomial in n and L if all intermediate values are normalized. Why does this not imply that Gaussian elimination runs in polynomial time without normalization of intermediate results?

3 Implement Gaussian elimination with floating point arithmetic. Find examples where the result of the floating point computation deviates widely from the exact result. Use the program of the first item to compute the exact result.

4.3 Floating Point Numbers

Floating point numbers are the computer science version of mathematics' real numbers. C++ offers single (type *float*) and double (type *double*) precision floating point numbers and LEDA offers in addition arbitrary precision floating point numbers (type *bigfloat*). Floating point arithmetic on most workstations adheres to the so-called IEEE floating point standard [IEE87], which we review briefly.

A floating point number consists of a sign s, a mantissa m, and an exponent e. In double format s has one bit, m consists of 52 bits $m_1, \ldots, m_{52}$, and e consists of the remaining 11 bits of a double word. The number represented by the triple (s, m, e) is defined as follows:

- e is interpreted as an integer in $[0 \,..\, 2^{11} - 1] = [0 \,..\, 2047]$.
- If $m_1 = \ldots = m_{52} = 0$ and $e = 0$ then the number is $+0$ or -0 depending on s.
- If $1 \le e \le 2046$ then the number is $s \cdot (1 + \sum_{1 \le i \le 52} m_i 2^{-i}) \cdot 2^{e-1023}$.

- If some m_i is non-zero and $e = 0$ then the number is $s \cdot \sum_{1 \le i \le 52} m_i 2^{-i} 2^{-1023}$. This is a so-called denormalized number.
- If all m_i are zero and $e = 2047$ then the number is $+\infty$ or $-\infty$ depending on s.
- In all other cases the triple represents NaN (= not a number).

The largest positive double (except for ∞) is $\texttt{MAXDOUBLE} = (2 - 2^{-52}) \cdot 2^{1023}$ and the smallest positive double is $\texttt{MINDOUBLE} = 2^{-52} \cdot 2^{-1023}$. Both constants are predefined in the systems file `value.h`. Arithmetic on floating point numbers is only approximate. For example,

```
float x = 123456789;
cout << (x + 1) - x;
```

will output 0 and not 1, the reason being that a nine-digit decimal number does not fit into a single precision floating point number. Thus, $cout \ll x$ will not reproduce 123456789. Although floating point arithmetic is inherently inexact, the IEEE standard guarantees that the result of any arithmetic operation is close to the exact result, usually as close as possible. Consider, for example, an addition $x + y$. If one of the arguments is NaN or the addition has no defined result, e.g., $-\infty + \infty$, then the result is NaN. Otherwise let z be the exact result. If $|z| > \texttt{MAXDOUBLE}$, as for example, in $\infty + (-5)$ or in $\texttt{MAXDOUBLE} + 1$, the result is $\pm\infty$, if $z < \texttt{MINDOUBLE}$ then the result is zero, and if $\texttt{MINDOUBLE} \le z \le \texttt{MAXDOUBLE}$ then the result is a floating point number $\tilde{z}$ which is closest to z. In particular,

$$|z - \tilde{z}| \le 2^{-53}|\tilde{z}|$$

since the error is at most 1 in the 53rd position after the binary point. The number $eps = 2^{-53}$ is frequently called the *precision* of double precision floating point arithmetic.

There is a rich body of literature on floating point arithmetic, see, for example, [DH91, Gol90, Gol91]. We do not pursue the properties of floats and doubles any further and turn to bigfloats instead.

The LEDA type *bigfloat* extends the built-in floating point types. The mantissa m and the exponent e of a bigfloat are arbitrary integers (type *integer*) and the number represented by a pair (m, e) is $m \cdot 2^e$. In addition, there are the special values ± 0 , $\pm\infty$, and NaN (= not a number). Arithmetic on bigfloats is governed by two parameters: the *mantissa length* and the *rounding mode*. Both parameters can either be set globally or for a single operation.

```
bigfloat::set_global_prec(212);
bigfloat::set_rounding_mode(TO_ZERO);
```

sets the mantissa length to 212 and the rounding mode to `TO_ZERO`. The arithmetic on bigfloats is defined as follows: let z be the exact result of an arithmetic operation. The mantissa of the result is obtained by rounding z to the prescribed number of binary places as dictated by the rounding mode. The available rounding modes are `TO_NEAREST` (round to the nearest representable number), `TO_P_INF` (round towards positive infinity), `TO_N_INF` (round towards negative infinity), `TO_ZERO` (round towards zero), `TO_INF` (round away from zero), and EXACT. For example[3], if the mantissa length is 3 and $z = 54371$ then the rounded value of z is

54400 if the rounding mode is TO_NEAREST or TO_P_INF or TO_INF, is

54300 if the rounding mode is TO_N_INF or TO_ZERO, and is

54371 if the rounding mode is EXACT.

The rounding mode EXACT applies only to addition, subtraction, and multiplication. In this mode the precision parameter is ignored and no rounding takes place. Since the exponents of bigfloats are arbitrary integers, arithmetic operations never underflow or overflow. However, exceptions may occur, e.g., division by zero or taking the square root of a negative number. They are handled according to the IEEE floating point standard, e.g., $5/0$ evaluates to ∞, $-5/0$ evaluates to $-\infty$, $\infty + 5$ evaluates to ∞, and $0/0$ evaluates to NaN.

The following inequality captures the essence of bigfloat arithmetic. If z is the exact result of an arithmetic operation and $\tilde{z}$ is the computed value then

$$|z - \tilde{z}| \leq 2^{-prec}|\tilde{z}|,$$

where *prec* is the mantissa length in use. With rounding mode TO_NEAREST the error bound is $2^{-prec-1}|\tilde{z}|$.

We illustrate bigfloats by a program that computes 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 $|z - e| \leq 2^{-m}$. Euler's number is defined as the value of the infinite series $\sum_{n \geq 0} 1/n!$. 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 $prec$ bits for still to be determined values of n_0 and $prec$, i.e., we execute the following program.

```
bigfloat::set_rounding_mode(TO_ZERO);
bigfloat::set_precision(prec);
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 *prec* 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,

$$|e - z_0| \leq |e - \textstyle\sum_{n<n_0} 1/n!| + |\textstyle\sum_{n<n_0} 1/n! - z_0|$$

[3] We use decimal notation instead of binary notation for this example.

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

The first term is certainly bounded by $2/n_0!$ since, for all $n \geq n_0$, $n! = n_0! \cdot (n_0+1) \cdot \ldots \cdot n \geq n_0! \cdot 2^{n-n_0}$ and hence $\sum_{n \geq n_0} 1/n! \leq 1/n_0! \cdot (1 + 1/2 + 1/4 + \ldots) \leq 2/n_0!$. 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 `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 $3 \cdot 2^{-prec}$. Thus

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

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 $2/n_0! \leq 2^{-m-2}$. 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 $n_0! \geq 2^{n_0}$ and the fact that n_0 is minimal with $2/n_0! \leq 2^{-m-2}$ we conclude $n_0 \leq m + 3$ and hence $6n_0 2^{-prec} \leq 6(m + 3) \cdot 2^{-prec} \leq 2^{-m-2}$ if $prec \geq 2m$; actually, $prec \geq m + \log(m + 3) + 5$ suffices. The following program implements this strategy and computes z_0 with $|e - z_0| \leq 2^{-m-1}$.

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 $z_0 \leq 3$ this will introduce an additional error of at most $3 \cdot 2^{-m-3} \leq 2^{-m-1}$. We conclude that the program below computes the desired approximation of Euler's number. This program is available as Euler_demo.

⟨*Euler_demo*⟩≡

```
main(){
int m;
⟨Euler: read m⟩
bigfloat::set_precision(2*m);
bigfloat::set_rounding_mode(TO_ZERO);
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,TO_NEAREST);
⟨Euler: output z⟩
}
```

Exercises for 4.3

1 Compute π with an error less than 2^{-200}.

2 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.

4.4 **Algebraic Numbers**

The data type *real* is LEDA's best approximation to mathematics' real numbers. It supports exact computation with k-th roots for arbitrary natural number k, the rational operators $+$, $-$, $*$, and $/$, and the comparison operators $==$, $!=$, $<$, $\le$, $\ge$, and $>$. Let us see a small example.

```
real x = (sqrt(17) - sqrt(12)) * (sqrt(17) + sqrt(12)) - 5;
cout << sign(x);
```

Note that the exact value of the expression defining x is 0. The distinctive feature of reals is that $sign(x)$ actually evaluates to zero. More generally, if E is any expression with integer operands and operators $+$, $-$, $*$, $/$, and function calls $sqrt(x)$ and $root(x,k)$ where x is a *real* and k is a positive integer then the data type real is able to determine the sign of E. We want to *stress that reals compute the sign of an expression in the mathematical sense* and not the sign of an approximation of an expression. This is in sharp contrast to the evaluation of an expression with floating point arithmetic. Floating point arithmetic incurs rounding error and hence, in general, cannot compute the sign of an expression correctly.

Why are we so concerned about the sign of expressions? The reason is that many programs contain conditional statements that branch on the sign of an expression and that such programs may go astray if the wrong decision about the sign is made. We give two examples, both arising in computational geometry. Further examples can be found in Section 8.6.

In the first example we consider the lines

$$l_1:\ y = 9833\cdot x/9454 \qquad \text{and} \qquad l_2:\ y = 9366\cdot x/9005.$$

Both lines pass through the origin and the slope of l_1 is slightly larger than the slope of l_2, see Figure 4.2. At $x = 9454\cdot 9005$ we have $y_1 = 9833\cdot 9005 = 9366\cdot 9454+1 = y_2+1$.

The following program runs through multiples of 0.001 between 0 and 1 and computes the corresponding y-values y_1 and y_2. It compares the two y-values and, if the outcome of the comparison is different than in the previous iteration, prints x together with the current outcome.

```
int last_comp = -1;
float a = 9833; float b = 9454;
float c = 9366; float d = 9005;
for (float x = 0; x < 1; x = x + 0.001)
```

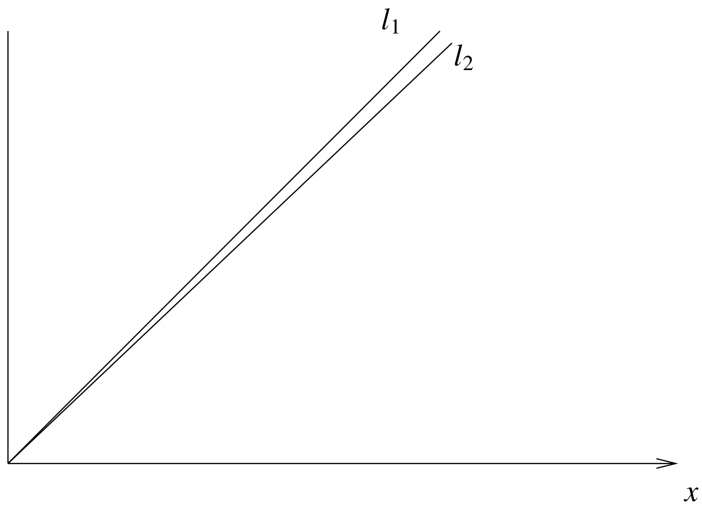

Figure 4.2 The lines l_1 and l_2: Both lines pass through the origin and l_1 is slightly steeper than l_2.

```
{ float y1 = a*x/b;   float y2 = c*x/d; // l1 is steeper
  int comp = (y1 < y2? -1 : (y1 == y2? 0 : +1));
  if (comp != last_comp)
  { cout <<"\nAt " << x << ": ";
    if (comp == -1) cout << "l1 is below l2";
    if (comp ==  0) cout << "l1 intersects l2";
    if (comp == +1) cout << "l1 is above l2";
  }
  last_comp = comp;
}
```

Clearly, we should expect the program to print:

```
At 0.000: l1 intersects l2
At 0.001: l1 is above l2
```

Well, the actual output on the first author's workstation contains the following lines[4]:

```
At 0.000: l1 intersects l2
At 0.003: l1 is above l2
At 0.004: l1 intersects l2
At 0.005: l1 is above l2
At 0.008: l1 intersects l2
At 0.009: l1 is below l2
...
At 0.993: l1 intersects l2
At 1.000: l1 is below l2
```

[4] If the program is run on the same author's notebook, it produces the correct result. The explanation for this behavior is that on the notebook double precision arithmetic is used to implement floats. According to the C++ standard floats must not offer more precision than doubles; they are not required to provide less. You may use the braided lines demo to find out how the program behaves on your machine.

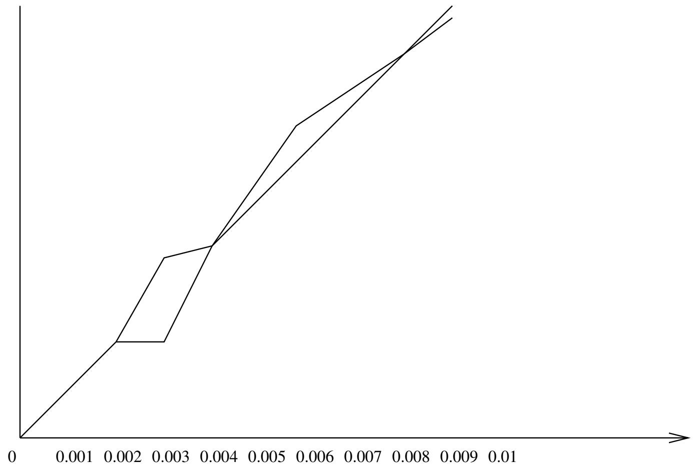

Figure 4.3 y_1 is equal to y_2 for $x = 0.001 \cdot i$ and i equal to 0, 1, and 2, is larger for i equal to 3, is equal for i equal to 4, is larger for i equal to 5, 6, and 7, is equal for i equal to 8, and is smaller for i equal to 9.

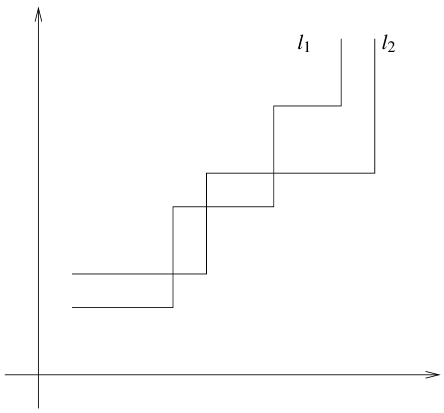

Figure 4.4 Lines as step functions and their multiple intersections.

We conclude that the lines intersect many times and are interlaced as shown in Figure 4.3. Observe that floating point arithmetic gives the wrong relationship between y_1 and y_2 not only for x close to zero but even for fairly large values of x. Thus the lines behave very differently from mathematical lines. Lyle Ramshaw coined the name *verzopfte Geraden (braided lines)* for the effect. Figure 4.4 explains the effect. The type *float* consists of only a finite number of values and hence a line is really a step function as shown in the figure. The width of the steps of our two lines l_1 and l_2 are distinct and hence the lines intersect.

The problem of braided lines is easily removed by the use of an exact number type; e.g., if *float* is replaced by *rational* in the program above, the output becomes what it should be:

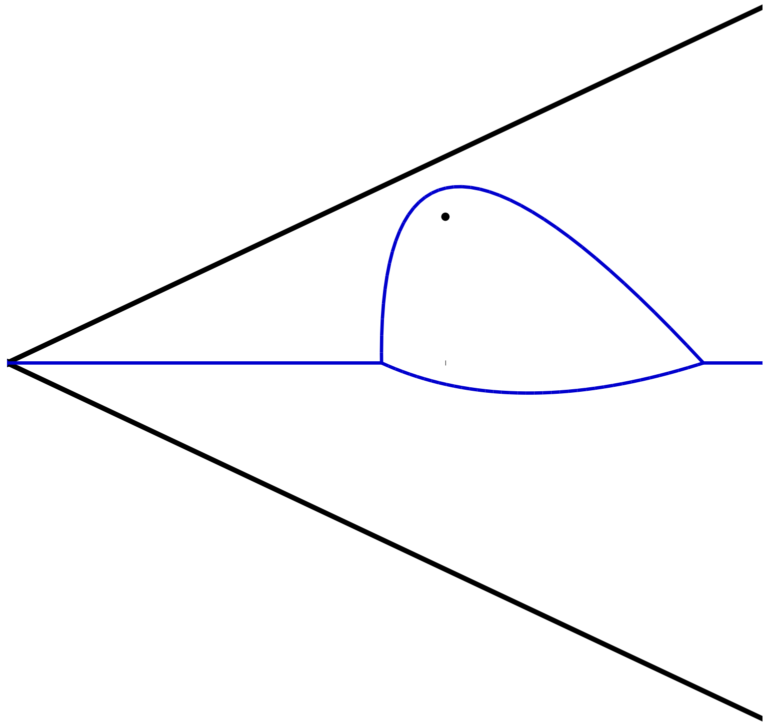

Figure 4.5 The Voronoi diagram of two lines l_1 and l_2 and a point p. The Voronoi diagram consists of parts of the angular bisector of l_1 and l_2 and of parts of the parabolas defined by p and l_1 and l_2, respectively. The Voronoi vertices are centers of circles passing through the point and touching the lines.

```
0/1: l1 intersects l2
1/1000: l1 is above l2
```

The second example goes beyond rational arithmetic and arises in the computation of Voronoi diagrams of line segments and points. Voronoi diagrams of line segments will be discussed in Section 9.5.5, and we assume for this paragraph that the reader has an intuitive understanding of Voronoi diagrams. For i, $1 \le i \le 2$, let $l_i : a_i x + b_i y + c_i = 0$ be a line in two-dimensional space and let $p = (0, 0)$ be the origin, cf. Figure 4.5. There are two circles passing through p and touching l_1 and l_2. These circles have centers $v_{1,2} = (x_{v_{1,2}}/z_v, y_{v_{1,2}}/z_v)$ where[5]

$$x_{v_{1,2}} = a_1 c_2 + a_2 c_1 \pm \sqrt{2 c_1 c_2 (\sqrt{N} + D)}$$

[5] The reader may compute these coordinates by solving the following equations for x_v/z_v and y_v/z_v.

$$\begin{aligned}(x_v/z_v)^2 + (y_v/z_v)^2 &= (a_1 x_v/z_v + b_1 y_v/z_v + c_1)^2/(a_1^2 + b_1^2)\\ (x_v/z_v)^2 + (y_v/z_v)^2 &= (a_2 x_v/z_v + b_2 y_v/z_v + c_2)^2/(a_2^2 + b_2^2)\end{aligned}$$

$$y_{v_{1,2}} = b_1c_2 + b_2c_1 \pm sign(S)\sqrt{2c_1c_2(\sqrt{N} - D)}$$

$$z_v = \sqrt{N} - a_1a_2 - b_1b_2$$

and $S = a_1b_2 + a_2b_1$, $N = (a_1^2 + b_1^2)(a_2^2 + b_2^2)$, and $D = a_1a_2 - b_1b_2$; in these expressions the $+$ in $\pm$ corresponds to v_1 and the $-$ corresponds to v_2. Consider now a third line l and let v be one of v_1 or v_2. The test of whether l intersects the circle centered at v is crucial for most algorithms computing Voronoi diagrams. Consider, for example, an incremental algorithm that adds the lines and points one by one and updates the diagram after every addition. Assume that such an algorithm has already constructed the diagram for p, l_1 and l_2 and next wants to add l. In the updated diagram the vertex v will not exist if l intersects the interior of the circle centered at v, v will exist and have degree four if l touches the circle centered at v, and v will exist and have the same incident edges if l does not intersect the circle centered at v. The question of whether l intersects, touches, or misses the circle centered at v is tantamount to comparing $dist(v, p)$ with $dist(v, l)$. We may also compare the squares of these numbers instead. The square of $dist(v, p)$ is $(x_v^2 + y_v^2)/z_v^2$ and the square of $dist(v, l)$ is $(ax_v/z_v + by_v/z_v + c)^2/(a^2 + b^2)$. In other words, we need to compute the sign of the expression

$$R = (ax_v + by_v + cz_v)^2 - (a^2 + b^2)(x_v^2 + y_v^2).$$

The following procedure takes inputs $a_1, b_1, \ldots, c$ and $pm \in \{-1, +1\}$ and performs this comparison; pm is used to select one of v_1 and v_2.

```
int INCIRCLE(integer a1, integer b1, integer c1, integer a2,
integer b2, integer c2, integer a, integer b, integer c, int pm)
{ real RN = sqrt((a1 * a1 + b1 * b1) * (a2 * a2 + b2 * b2));
  real A  = a1 * c2 + a2 * c1;
  real B  = b1 * c2 + b2 * c1;
  real C  = 2 * c1 * c2;
  real D  = a1 * a2 - b1 * b2;
  real S  = a1 * b2 + a2 * b1;
  real xv = A + pm * sqrt(C * (RN + D));
  real yv = B + pm * sign(S) * sqrt(C * (RN - D));
  real zv = RN - (a1 * a2 + b1 * b2);
  real P  = a * xv + b * yv + c * zv;
  real R  = P * P - (a * a + b * b) * (xv * xv + yv * yv);
  return sign(R);
}
```

How do reals work? The sign computation is based on the concept of a *separation bound*. A separation bound for an expression E is an *easily computable* number $sep(E)$ such that

$$val(E) \neq 0 \text{ implies } |val(E)| \geq sep(E),$$

where $val(E)$ denotes the value of E. Thus $|val(E)| < sep(E)$ implies $val(E) = 0$. Given a separation bound there is a simple strategy to determine the sign of $val(E)$:

- Compute an approximation A of $val(E)$ with $|A - val(E)| < sep(E)/2$ by evaluating E with *bigfloat* arithmetic with sufficient mantissa length. The required mantissa length can be determined by an error analysis in the same way, as we determined the mantissa length required for the computation of Euler's number with an error less than 2^{-m} in the preceding section. We stress that this error analysis is automated in the data type *real* and is invisible to the user.
- If $|A| \geq sep(E)/2$ then return the sign of A and if $|A| < sep(E)/2$ then return zero.

The correctness of this approach can be seen as follows:

If $|A| \geq sep(E)/2$ then $|A - val(E)| < sep(E)/2$ implies that $val(E)$ and A have the same sign.

If $|A| < sep(E)/2$ then $|A - val(E)| < sep(E)/2$ implies $|val(E)| < sep(E)$. Thus, $val(E) = 0$ by the definition of a separation bound.

Next, we give the separation bound that is used in LEDA. First, we need to define precisely what we mean by an expression. For simplicity, we deal only with expressions without divisions, although *reals* also handle divisions. An expression E is an acyclic directed graph (dag) in which each node has indegree at most two, in which each node of indegree 0 is labeled by a non-negative integer, each node of indegree 1 is labeled either by $-$ (unary minus) or by $root_k$ for some natural number k, and each node of indegree 2 is labeled by either a $+$ or a $*$. Figure 4.6 shows an expression. We define the *degree* $deg(E)$ of E as the product of the k's over all nodes labeled by root operations. The expression of Figure 4.6 has degree 4. We define the *bound* $b(E)$ of E as the value of the expression $\hat{E}$ which is obtained from E by removing all nodes labeled with a unary minus and connecting their input node directly to their outputs. In our example, we have $b(E) = (\sqrt{17} + \sqrt{12})(\sqrt{17} + \sqrt{12}) + 5$.

Theorem 1 ([BRMS97]) *Let E be an expression. Then $val(E) \leq b(E)$ and either*

$$val(E) = 0 \quad or \quad |val(E)| \geq b(E)^{1-deg(E)}.$$

We give a proof of a special case. Assume that A, B, and C are natural numbers. How close to zero can $A\sqrt{B} - C$ be, if non-zero? We have

$$\begin{aligned} |A\sqrt{B} - C| &= |A\sqrt{B} - C| \cdot (A\sqrt{B} + C)/(A\sqrt{B} + C) \\ &= |A^2 B - C^2|/(A\sqrt{B} + C) \\ &\geq 1/(A\sqrt{B} + C), \end{aligned}$$

where the last inequality follows from the assumption that the value of our expression is different from zero and from the fact that $A^2B - C^2$ is an integer. The expression above has degree 2 and its b-value is equal to $A\sqrt{B} + C$. Thus, the derived bound corresponds precisely to the statement of the theorem.

It is worthwhile to restate the theorem in terms of the binary representation of $val(E)$. Let $L = \log b(E)$. Then $|val(E)| \leq 2^L$ and, if $val(E) \neq 0$, $|val(E)| \geq 2^{L\cdot(1-deg(E))}$. Thus, if $val(E) \neq 0$, then the binary representation of $val(E)$ either contains a non-zero digit in

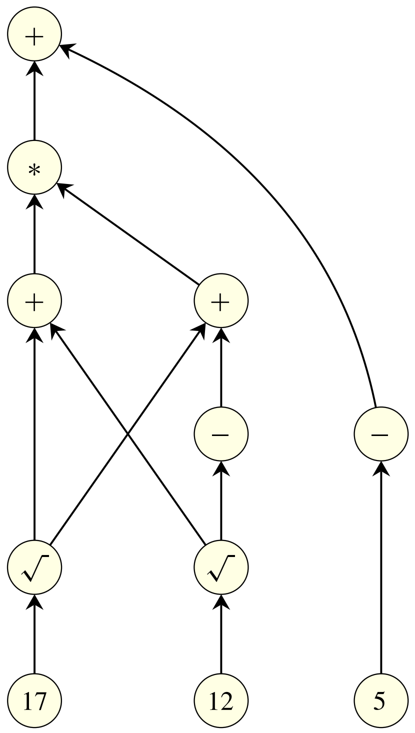

Figure 4.6 An expression dag E. The expression has degree 4 and computes $(\sqrt{17}+\sqrt{12})\cdot(\sqrt{17}-\sqrt{12})-5$.

the L digits before the binary point or a non-zero digit in the first $(deg(E)-1)\cdot L$ digits after the binary point. Conversely, if all of these digits are zero then $val(E)$ is zero. In the sequel we will rephrase this statement as: It suffices to inspect the first $deg(E)\cdot L$ bits of the binary representation of $val(E)$.

We give two applications of the theorem above. They are illustrated by the two real number demos, respectively.

First, let x be an arbitrary integer and consider the expression

$$E_1 = (\sqrt{x+5}+\sqrt{x})(\sqrt{x+5}-\sqrt{x})-5.$$

Then $deg(E_1) = 4$ and $b(E_1) < 4(x+5)+5$. Let $L_1 = \log(4(x+5)+5)$. By the theorem above it therefore suffices to inspect the first $4L_1$ bits of the binary representation of $val(E_1)$ in order to determine its sign. So if x has 100 binary digits it certainly suffices to inspect 412 digits of $val(E_1)$. This is illustrated by the program below. It asks for an integer L and then constructs a random integer x with L decimal digits. It then computes the signs of E_1+5 and E_1.

L	80	160	320	640	1280	2560	5120
A	0.01	0.01	0.03	0.05	0.14	0.41	1.47
B	0.04	0.07	0.21	0.66	2.24	8.03	29.73

Table 4.2 The running times for computing the signs of $A = (\sqrt{x+5} + \sqrt{x})(\sqrt{x+5} - \sqrt{x})$ and $B = A - 5$ for x being a random integer with L decimal digits. Note that the time for computing the sign of A is much smaller than the time for computing the sign of B. This reflects the fact that a crude approximation of A allows us to conclude that A is positive and that about $4L$ digits of B need to be computed in order to allow the conclusion that B is zero. You may perform your own experiments by calling the first real number demo.

⟨*real_demo1*⟩≡

```
⟨real demo1: read L⟩
integer x = 0;
while (L > 0)
{ x = x*10 + rand_int(0,9);
  L--;
}
float T = used_time();
real X = x;
real SX = sqrt(X);
real SXP = sqrt(X+5);
real A = (SXP + SX) * (SXP - SX);
real B = A - 5;
int A_sign = A.sign(); float TA = used_time(T);
int B_sign = B.sign(); float TB = used_time(T);
⟨real demo1: output signs and report running times⟩
```

Table 4.2 shows the running times of this program for $L = 80$, 160, 320, and so on.

Next, consider the expression

$$E_2 = (2^{2^k} + 1)^{2^{-k}} - 2,$$

i.e., the number 2 is squared k times, 1 is added , square roots are taken k times, and finally 2 is subtracted. This yields a number slightly above 0. In fact[6],

$$\begin{aligned} val(E_2) &= (2^{2^k} + 1)^{2^{-k}} - 2 \;=\; 2((1 + 2^{-2^k})^{2^{-k}} - 1) \\ &= 2(\exp(2^{-k} \ln(1 + 2^{-2^k})) - 1) \;\approx\; 2(\exp(2^{-k} 2^{-2^k}) - 1) \\ &\approx 2(1 + 2^{-k} 2^{-2^k} - 1) \;=\; 2^{1-k-2^k}, \end{aligned}$$

i.e., the first non-zero bit in the binary expansion of $val(E_2)$ is about $k+2^k$ positions after the binary point. What does the theorem above say? We have $deg(E_2) = 2^k$ and $b(E_2) \le 5$ and hence by the theorem it suffices to inspect the first $2^k \log 5$ bits of the binary representation

[6] We use the estimates $\ln(1 + x) \approx x$ and $e^x \approx 1 + x$ for x close to zero.

of $val(E_2)$. That's an overestimate by about a factor of two. The following program chunk illustrates this example. It asks for an integer k and then computes the sign of the expression E_2. It also shows the *bigfloat* approximation of E_2 that is computed in the sign computation.

⟨*real_demo2*⟩≡
```
int k = I.read_int("k = ");
float T = used_time();
real E = 2;
int i;
for (i = 0; i < k; i++)  { E = E*E; }
E = E + 1;
for (i = 0; i < k; i++) { E = sqrt(E); }
E = E - 2;
I.write_demo("The sign of E is ",E.sign(),".");
I.write_demo("This took ",used_time(T)," seconds.");
I.write_demo("An approximation of E: " + to_string(E.to_bigfloat()));
```

We close this section with a brief discussion of the implementation of reals. The data type *real* stores objects of type real by their expression dags, i.e., every operation on reals adds a node to the expression dag and records the arithmetic operation to be performed at the node and the inputs to the node. Thus the dag of Figure 4.6 is built for the expression $(\sqrt{17}+\sqrt{12})(\sqrt{17}-\sqrt{12})-5$. Whenever the sign of a real number has to be determined, a separation bound is computed as described in Theorem 1 and then a bigfloat computation is performed to determine the sign.

We sketch how the bigfloat computation is performed; for details we refer the reader to [BMS96].We set a parameter l to some small integer and compute an approximation A of $val(E)$ with $|A - val(E)| < 2^{-l}$. In order to compute such an approximation an error analysis along the lines of the preceding section is performed (this is fully automated) and then a bigfloat computation with the appropriate mantissa length is performed. If $|A| \geq 2 \cdot 2^{-l}$ then $val(E)$ and A have the same sign and we may return the sign of A. If $|A| < 2 \cdot 2^{-l}$ we double l and repeat. We continue in this fashion until $2^{-l} \leq sep(E)/2$, where $sep(E)$ is the separation bound. Table 4.2 illustrates the effect of this optimization: For the expression A a crude approximation allows us to decide the sign and hence $sign(A)$ is computed quickly, however, for expression B one has to go all the way to the quality of approximation prescribed by the separation bound.

We close with a warning. Reals are not a panacea. Although they allow in principle to compute the sign of any expression involving addition, subtraction, multiplication, division, and arbitrary roots, you may have to wait a long time for the answer when the expression is complex. The paper [**?**] discusses the use of *reals* in geometric computations.

Exercises for 4.4

1 Compute the sign of $E = (2^{2^k}+1)^{2^{-k}} - 2$ for different values of k. You may use program real_demo2 for this purpose. Don't be too ambitious. Try to predict the growth rate of the running time before performing the experiment.

2 Let $E = E_1 + E_2$ and assume that you want an approximation A of $val(E)$ such that $|val(E) - A| \leq \varepsilon$. Determine ε_1 and ε_2 and a precision *prec* such that computation of bigfloat approximations A_i of $val(E_i)$ with an error $|A_i - val(E_i)| \leq \varepsilon_i$ and summation of A_1 and A_2 with precision *prec* yields the desired approximation A of $val(E)$.

3 As above for $E = E_1 \cdot E_2$, $E = E_1/E_2$, and $E = \sqrt{E_1}$. Solutions to exercises 2. and 3. can be found in [BMS96].

4 Let p_1 and p_2 be two points in the plane, let l be a line, and consider the circle passing through p_1 and p_2 and touching l. Write a procedure that determines the position of a third point p_3 with respect to this circle.

4.5 Vectors and Matrices

Vectors and matrices are one- and two-dimensional arrays of numbers, respectively. Let n and m be integers. An n-dimensional vector v is a one-dimensional arrangement of n variables of some number type N; the variables are indexed from 0 to $n - 1$ and $v[i]$ denotes the variable with index i. An $n \times m$ matrix M is a two-dimensional arrangement of $n \cdot m$ variables of some number type N; the variables are indexed by pairs (i, j) with $0 \leq i \leq n - 1$ and $0 \leq j \leq m - 1$. We use $M(i, j)$ to denote the variable indexed by i and j and call n and m the number of rows and columns of M, respectively. Observe that as for two-dimensional arrays we use round brackets for the subscript operator in matrices. We have currently vectors and matrices with entries of type *double* (types *vector* and *matrix*) and type *integer* (types *integer_vector* and *integer_matrix*). Vectors and matrices over an arbitrary number type are part of the LEP for higher-dimensional geometry. We use the latter types in all our examples. The definitions

```
integer_vector v(m);
integer_matrix M(n,m);
```

define an m-vector v and an $n \times m$-matrix M, respectively. All entries of v and M are initialized to zero. The following procedure multiplies a matrix M by a vector v.

```
integer_vector integer_matrix::operator*(const integer_matrix& M,
                                         const integer_vector& v)
{ int n = M.dim1(); // # of rows of M
  int m = M.dim2(); // # of columns of M
  if (m != v.dim()) error_handler(1, "incompatible dimensions");
  integer_vector result(n);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) result[i] += M(i,j) * v[j];
  return result;
}
```

In the context of

```
integer_vector v(5);
integer_vector r;       // a 0 - dimensional vector
integer_matrix M(3, 5);
```

we may now write

```
r = M * v.
```

Note that we defined r as an empty vector. The assignment $r = M * v$ assigns the result of the multiplication $M * v$ to r. This involves allocation of memory (for three variables of type *integer*) and component-wise assignment. Vectors are internally represented as a pair consisting of an int *dim*, containing the dimension of the vector, and a pointer v to a C++ array containing the components of the vector. The code for the assignment operator is as follows:

```
integer_vector& integer_vector::operator=(const integer_vector& vec)
{ if (dim != vec.dim())
  { /* this vector does not yet have the right dimension */
    delete v;
    dim = vec.dim();
    v = new integer[dim];
  }
  for (int i = 0; i < dim; i++) v[i] = vec[i];
  return *this;
}
```

Vectors and matrices are similar to one- and two-dimensional C++ arrays of numbers, respectively. The main differences are as follows:

- Vectors and matrices know their dimension(s). Assignment is component-wise assignment. It allocates space automatically.
- Vectors and matrices check whether indices are legal. The checks can be turned off.
- Vectors and matrices are somewhat slower than their C++ counterparts.
- Vectors and matrices offer a large number of operations of linear algebra.

The basic operations of linear algebra are vector and matrix addition and multiplication, and multiplication by a scalar. For example, $M+N$ denotes the component-wise addition of two matrices M and N, $M * N$ denotes matrix multiplication, $M * v$ denotes matrix-vector product, $v * w$ is the scalar product of two vectors, and $v * 5$ multiplies each entry of v by the scalar 5.

We turn to the more advanced functions of linear algebra. Let M be an $n \times m$ integer matrix and let b be an n integer vector. Let x be an integer vector and let D be an integer variable. The call

```
linear_solver(M,b,x,D);
```

returns true if the linear system $M \cdot z = b$ has a solution and returns false otherwise. If the system is solvable then the vector $(1/D) \cdot x$ is a solution of the system. Why do we return the solution in this strange format? The solution vector of the system $M \cdot z = b$ has rational entries. We provide a common denominator in D and the numerator in x. For example, the system

$$\begin{aligned} 3z_0 + z_1 &= 5 \\ z_0 + z_1 &= 2 \end{aligned}$$

has the solution $z = (3/2, 1/2)$. We return this solution as $x = (3, 1)$ and $D = 2$.

The main use of linear algebra within LEDA is the exact implementation of geometric primitives; e.g., we solve a linear system to determine the equation of a hyperplane through a set of points and we compute a determinant to determine the orientation of a sequence of points. We use matrices and vectors over integers for that purpose. We hardly use vectors and matrices over doubles within LEDA and therefore have not optimized the robustness of our linear system solver. We do not recommend to use our procedures for serious numerical analysis. Much better codes are available in the numerical analysis literature. A good source of codes is the book [FPTV88].

A linear system $M \cdot z = b$ may have more than one solution, may have exactly one solution, or no solution at all. The call *linear_solver*(M, b, x, D, s_vecs, c) gives complete information about the solution space of the system $M \cdot z = b$:

- If the system is unsolvable then c is an n-vector such that $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$, i.e., c specifies a linear combination of the equations such that the left-hand side of the resulting equation is identically zero and the right-hand side is non-zero. For example, for the system

$$\begin{aligned} z_0 + z_1 &= 5 \\ 2z_0 + 2z_1 &= 4 \end{aligned}$$

  the vector $c = (-2, 1)^T$ *proves* that the system is unsolvable.

- If the system is solvable then $(1/D) \cdot x$ is a solution and *s_vecs* is an $m \times d$ matrix for some d whose columns span the solution space of the corresponding homogeneous system $M \cdot z = 0$. Let col_j denote the j-th column of *s_vecs*. Then any solution to $M \cdot z = b$ can be written as

$$(1/D) \cdot x + \sum_{o \le j < d} \lambda_j \cdot col_j$$

  for some reals λ_j, $0 \le j < d$. You may extract the j-th column of *s_vecs* by *s_vecs.col*(j).

The rank of a matrix is the maximal number of linearly independent rows (or columns). The call

```
rank(M);
```

returns the rank of M.

From now on we assume that M is a square matrix, i.e., an $n \times n$ matrix for some n. A square matrix is called *invertible* or *non-singular* if there is a matrix N such that $M \cdot N = N \cdot M = I$, where I is the $n \times n$ identity matrix; the matrix N is called the inverse of M and is usually denoted by M^{-1}. A matrix without an inverse is called *singular*. A matrix is singular if and only if its determinant is equal to zero. The call

```
integer D = determinant(M);
```

returns the determinant of M. The inverse of an integer matrix has, in general, rational entries.

```
integer_matrix N = inverse(M,D);
```

assigns a common denominator of the entries of the inverse to D and returns the matrix of numerators in N, i.e, $(1/D) \cdot N$ is the inverse of M. The function *inverse* requires that M is non-singular and hence should only be used if M is known to be non-singular. The call

```
inverse(M,N,D,c);
```

returns true if M has an inverse and false otherwise. In the former case $(1/D) \cdot N$ is the inverse of M and in the latter case c is a non-zero vector with $c^T \cdot M = 0$. Note that such a vector proves that M is singular.

The LU-decomposition of a matrix is the decomposition as a product of a lower and an upper diagonal matrix.

```
LU_decomposition(M,L,U,q);
```

computes a lower diagonal matrix L, an upper diagonal matrix U, and a permutation q of $[0 \,..\, n-1]$ (represented as an *array*<*int*>) such that for all i, $0 \leq i < n$, the $q[i]$-th column of $L \cdot M$ is equal to the i-th column of U.

Exercises for 4.5

1 Write a procedure that determines whether a homogeneous linear system has a non-trivial solution.

2 Write a function that computes the equation of a hyperplane passing through a given set of d points in d-dimensional Euclidean space.

Bibliography

[BMS96] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, Saarbrücken, January 1996.

[BRMS97] Ch. Burnikel, R.Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots(ps). In *Proc. SODA 97*, pages 702–709, 1997.

[DH91] P. Deuflhard and A. Hohmann. *Numerische Mathematik: Eine algorithmisch orientierte Einführung*. Walter de Gruyter, 1991.

[FPTV88] Brian P. Flannery, William H. Press, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1988.

[Gol90] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1990.

[Gol91] David Goldberg. Corrigendum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 23(3):413–413, September 1991.

[IEE87] IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN Notices*, 2:9–25, 1987.

[Kar63] A. Karatsuba. *Soviet Physics-Doklady*, 7:595–596, 1963.

Index