Computation of the scalar product of the two vectors $X = (x^k)$ and $Y = (y^k)$ is just the accumulation of all the products in (9.7.4) for k from 1 to n:

$$X \cdot Y = \sum_{k=1}^{n} x^k \cdot y^k = \sum_{k=1}^{n} \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i^k \cdot y_j^k .$$

This shows that the scalar product of the two multiple precision vectors X and Y can be computed by accumulating products of floating-point numbers in a single complete variable cv. The result is a multiple precision number $z = \sum_{i=1}^{n_z} z_i$. It is obtained by conversion of the complete register contents into the multiple precision number z by:

for $i := 1$ to n_z do
  $z_i := \text{chop}(cv)$
  $cv := cv - z_i$

Similar considerations hold for the computation of two multiple precision matrices or for the computation of the defect of a system of linear equations with multiple precision data. Of course, the formulas for these computations are getting more and more complicated. But the user does not have to be concerned with these. By operator overloading the work is done by the computer automatically.

The key operation for all these processes is a fast and exact scalar product. Quadruple precision arithmetic is not a substitute for it.

At several occasions in this section rounding towards zero is applied where rounding to nearest could have been used instead. The reason for this is that rounding towards zero is simpler and faster in general than rounding to nearest.

9.7.2 Multiple precision interval arithmetic

Definition 9.8. Let x_i, $i = 1(1)n$, $n \geq 0$, be floating-point numbers and $X = [x_{\text{low}}, x_{\text{high}}]$ an interval with floating-point bounds x_{low} and x_{high}. Then an element of the form

$$x = \sum_{i=1}^{n} x_i + X \tag{9.7.5}$$

is called a *long interval* of *length* n. The x_i are called the components of x and X is called the *interval component*. ■

In (9.7.5) n is permitted to be zero. Then the sum in (9.7.5) is empty and $x = X$ is just an interval with standard floating-point bounds.

In the representation (9.7.5) of a long interval it is desirable that the components do not overlap. The following operations for long intervals are written so that they produce results with this property.

Arithmetic operations for long intervals are defined as usual in interval arithmetic:

Definition 9.9. Let x and y be long intervals, then

$$x \circ y := \{\xi \circ \eta | \xi \in x \wedge \eta \in y\}, \text{ for } \circ \in \{+, -, \cdot, /\},$$

with $0 \notin y$ for $\circ = /$. ■

Of course, in general, this theoretical result is not representable on the computer. Here the result must be a long interval again. We do not, however, require that it is the least enclosing long interval of some prescribed length. But we must require that the computed long interval z is a superset of the result defined in Definition 9.9: $x \circ y \subseteq z$. Not to require optimality of the result gives room for a compromise between tightness of the enclosure and the efficiency of the implementation.

Negation.

$$-x = \sum_{i=1}^{n_x} (-x_i) + [-x_{\text{high}}, -x_{\text{low}}].$$

Let now x, y, and z be three long intervals:

$$x = \sum_{i=1}^{n_x} x_i + [x_{\text{low}}, x_{\text{high}}], \quad y = \sum_{i=1}^{n_y} y_i + [y_{\text{low}}, y_{\text{high}}], \quad z = \sum_{i=1}^{n_z} z_i + [z_{\text{low}}, z_{\text{high}}].$$

Addition and subtraction of two long intervals x and y simply add and subtract the lower and higher bounds of x and y into two complete registers which we call *lo* and *hi*. Their contents finally have to be converted into a long interval z. This conversion routine will be discussed after the description of the operations of addition and subtraction. In the following algorithms the symbols $+$, $-$, and $\cdot$ again denote the operations for real numbers.

Addition.

$$\begin{aligned} hi &:= \sum_{i=1}^{n_x} x_i + \sum_{i=1}^{n_y} y_i \\ lo &:= hi + x_{\text{low}} + y_{\text{low}} \\ hi &:= hi + x_{\text{high}} + y_{\text{high}} \\ z &:= \text{convert}(lo, hi, n_z). \end{aligned}$$

Subtraction.

$$\begin{aligned} hi &:= \sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} y_i \\ lo &:= hi + x_{\text{low}} - y_{\text{high}} \\ hi &:= hi + x_{\text{high}} - y_{\text{low}} \\ z &:= \text{convert}(lo, hi, n_z). \end{aligned}$$

Conversion.

```
convert(lo, hi, n_z)
  stop := false
  i := 0
  repeat i := i + 1
    z_low := chop(lo)
    z_high := chop(hi)
    if z_low = z_high then z_i := z_low
      lo := lo − z_i
      hi := hi − z_i
    else z_i := 0
      stop := true
  until stop or i = n_z
  for i := i + 1 to n_z do z_i := 0
    z_low := ▽lo
    z_high := △hi
```

This routine reads successive numbers z_{low} and z_{high} from the complete registers and, as long as they are equal and the length n_z has not yet been reached, they are assigned to z_i and subtracted from the complete registers. If z_{low} and z_{high} are different or the length n_z for the result z is reached, then the remaining values in the complete registers are converted to the interval component Z of z by appropriate rounding. If some of the z_i are not yet defined, they are set to zero.

The conversion routine has the property that the real components of z do not overlap.

Multiplication. Multiplication can be implemented in various ways yielding different results because of the subdistributivity law of interval arithmetic. Thus we have:

$$\begin{aligned} x \cdot y &= \Big(\sum_{i=1}^{n_x} x_i + X\Big)\Big(\sum_{j=1}^{n_y} y_j + Y\Big) \\ &\subseteq \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i \cdot y_j + X\sum_{j=1}^{n_y} y_j + Y\sum_{i=1}^{n_x} x_i + XY \qquad (9.7.6) \\ &\subseteq \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i \cdot y_j + \sum_{j=1}^{n_y} Xy_j + \sum_{i=1}^{n_x} Yx_i + XY. \qquad (9.7.7) \end{aligned}$$

This seems to suggest that better results can be obtained from using the second line in (9.7.6) than from the third line. However, to compute the products $X\sum_{i=1}^{n_y} y_i$ and $Y\sum_{i=1}^{n_x} x_i$ we first have to round the sums to machine intervals. As a consequence of these additional roundings, the second line in (9.7.6) yields coarser enclosures than the third line. Therefore, we use line three for the multiplication algorithm.

Again, *lo* and *hi* are two complete registers, x_i, y_i are reals, X, Y are intervals, and z is the resulting long interval. The multiplication routine is as follows:

$$\begin{aligned} lo &:= \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} x_i \cdot y_j \\ hi &:= lo \\ [lo, hi] &:= [lo, hi] + \sum_{j=1}^{n_y} X y_j + \sum_{i=1}^{n_x} Y x_i + X \cdot Y \\ z &:= \mathrm{convert}(lo, hi, n_z). \end{aligned}$$

Here again all accumulations in *lo* and *hi* are to be done without any intermediate roundings.

Division. For division, again an iterative algorithm is applied. It computes the n_z real components z_i of the quotient x/y successively.

To compute the approximation $\sum_{i=1}^{n_z} z_i$, we start with

$$z_1 := \square m(x) \boxslash \square m(y).$$

Here $m(x)$ and $m(y)$ represent points selected within x and y respectively, the midpoints for instance. The rounding to a floating-point number by the rounding towards zero is denoted by $\square$ and $\boxslash$ means floating-point division.

Now the components z_i, $i = 1(1)n_z$ of z are computed by the same formula as for a long real arithmetic:

$$z_{k+1} := \square \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{k} y_i z_j \right) \boxslash (\square m(y)). \tag{9.7.8}$$

Here the numerator is computed exactly in a complete register and then rounded towards zero into a floating-point number. Finally a floating-point division is performed.

This iteration again guarantees that the z_i do not overlap.

Now the interval component Z of the result z is computed as

$$Z := \diamondsuit \left(\sum_{i=1}^{n_x} x_i - \sum_{i=1}^{n_y} \sum_{j=1}^{n_z} y_i z_j + X - \sum_{j=1}^{n_z} Y z_j \right) \diamondsuit (\diamondsuit y), \tag{9.7.9}$$

where $\diamondsuit$ denotes the rounding to a floating-point interval and $\diamondsuit$ denotes the division of two floating-point intervals.

It is not difficult to see that $z = \sum_{i=1}^{n_z} z_i + Z$ is a superset of the exact range $\{\xi \circ \eta \mid \xi \in x \wedge \eta \in y\}$. For $\alpha \in X$ and $\beta \in Y$ we have the identity

$$\frac{\sum_i^{n_x} x_i + \alpha}{\sum_i^{n_y} y_i + \beta} = \sum_j^{n_z} z_j + \frac{\sum_i^{n_x} x_i + \alpha - \sum_i^{n_y} \sum_j^{n_z} y_i z_j - \sum_j^{n_z} z_j \beta}{\sum_i^{n_y} y_i + \beta}.$$

An interval evaluation of this expression for $\alpha \in X$ and $\beta \in Y$ shows immediately that the exact range x/y is contained in $\sum_{i=1}^{n_z} z_i + Z$ as computed by (9.7.8) and (9.7.9).

This leads to the following algorithm. Therein x_i, y_i, z_i, and y_m are floating-point reals, X, Y, and Z are floating-point intervals. Rounding towards zero into a floating-point number is denoted by $\square$ and $\diamond$ denotes the rounding to a floating-point interval.

$lo := \sum_{i=1}^{n_x} x_i$
$my := \square m(y)$
$z_1 := (\square lo) \boxslash my$
for $k := 2$ to n_z do
  $lo := lo - \sum_{i=1}^{n_y} y_i z_{k-1}$
  $z_k := (\square lo) \boxslash my$
$lo := lo - \sum_{i=1}^{n_y} y_i z_{n_z}$
$hi := lo$
$Z := \diamond([lo, hi] + X - \sum_{i=1}^{n_z} Y z_i) \diamondslash (\diamond y)$.

In this algorithm the double sum in (9.7.8) and (9.7.9) is accumulated in the complete register lo as long as the z_k are computed. The final value in lo is then used in the computation of the interval part Z. Thus the amount of work is reduced to a minimum.

Scalar Product. We leave it to the reader to derive formulas for the computation of the scalar product of two vectors with long interval components. We mention, however, that this is not necessary at all. By operator overloading the computer solves this problem automatically. What is needed are two complete registers and a fast and exact scalar product for floating-point numbers.

Square Root. An algorithm for the square root can be obtained analogously as in the case of division. It computes the z_i, $i = 1(1)n_z$ of the approximation part iteratively:

$$z_1 := \sqrt{\square x},$$
$$z_{k+1} := \square\left(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{k} z_i z_j\right)/(2z_1). \tag{9.7.10}$$

This guarantees that the z_i do not overlap since in the numerator of (9.7.10) the defect of the approximation $\sum_{i=1}^{n_z} z_i$ is computed with one rounding only. Now the interval part Z is computed as

$$Z := \frac{\diamond\left(\sum_{i=1}^{n_x} x_i - \sum_{i,j=1}^{n_z} z_i z_j + X\right)}{\sqrt{\diamond x} + \diamond \sum_{i=1}^{n_z} z_i}. \tag{9.7.11}$$

As in the case of division, it is easy to see that $\sum_{i=1}^{n_z} z_i + Z$ as computed by (9.7.10) and (9.7.11) is a superset of the exact range $\{\sqrt{\xi} \mid \xi \in x\}$; in fact, for all $\gamma \in X$ we have the identity:

$$\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} = \sum_{j=1}^{n_z} z_j + \frac{\sum_{i=1}^{n_x} x_i + \gamma - \sum_{i,j=1}^{n_z} z_i z_j}{\sqrt{\sum_{i=1}^{n_x} x_i + \gamma} + \sum_{j=1}^{n_z} z_j}. \tag{9.7.12}$$

This leads to the following algorithm for the computation of the square root:

$lo := \sum_{i=1}^{n_x} x_i$
$z_1 := \sqrt{\square\, lo}$
for $k := 2$ to n_z do
  $lo := lo - 2\sum_{j=1}^{k-2} z_j z_{k-1} - z_{k-1} z_{k-1}$
  $z_k := (\square\, lo)/(2z_1)$
$lo := lo - 2\sum_{j=1}^{n_z-1} z_j z_{n_z} - z_{n_z} z_{n_z}$
$hi := lo$
$Z := \diamond([lo, hi] + X)/(\sqrt{\diamond x} + \diamond \sum_{j=1}^{n_z} z_j)$.

To allow easy application of the long interval arithmetic just described a few additional operations should be supplied such as computation of the infimum, the supremum, the diameter, and the midpoint. Also elementary functions can be and have been implemented for long intervals. They may already make use of the arithmetic for long intervals.

9.7.3 Applications

We now briefly sketch a few applications of multiple precision interval arithmetic. We restrict the discussion to problem classes which have already been dealt with in this chapter. We assume that the floating-point inputs to an algorithm are exact. Imprecise data should be brought into the computer as intervals as accurately as possible, possibly as long intervals. It should be clear that in such a case the result is a set, and if the algorithm is unstable, this set may well be large. Even the best arithmetic can only compute bounds for this set. These bounds may not look very accurate even if they may be so.

Among the first problems that have been solved to very high and guaranteed accuracy were systems of linear equations by S. M. Rump [557]. The method was then continually extended to other problem classes. We consider a system of linear equations $A \cdot x = b$. Let x_1 be an approximate solution and $e_1 := x^* - x_1$ be the error to the exact solution x^*. Then e_1 is the solution of the system

$$A \cdot e_1 = b - Ax_1. \tag{9.7.13}$$

If we compute an interval enclosure X_1 of e_1, we have an enclosure of x^* by a long interval: $x^* \in x_1 + X_1$. This method can now be iterated. With a new approximate solution $x_2 := m(x_1 + X_1)$, where m denotes the midpoint of the interval $x_1 + X_1$, a second error e_2 can be computed by

$$A \cdot e_2 = b - Ax_2.$$

An enclosure X_2 of e_2 leads to a new enclosure of x^*:

$$x^* \in x_2 + X_2.$$

Essential for success of this method is the fact that the defect $b - A \cdot x_i$ of the approximate solution x_i can be computed to full accuracy by the exact scalar product.

This method of iterated defect correction can also be applied to compute with very high accuracy enclosures of arithmetic expressions or of polynomials. An enclosure of the solution is obtained as a long interval.

The methods just discussed can also be applied to problems where the initial data themselves are long intervals. See [423].

The method for the evaluation of polynomials with long interval coefficients allows additional applications. It can be applied, for instance, to evaluate higher dimensional polynomials and to represent the result as a long interval. To avoid too many indices we sketch the method for the two-dimensional case. The independent variables are denoted by x and y. Let the polynomial of degree n in x and m in y be

$$p(x, y) = \sum_{j=0}^{m} \sum_{i=0}^{n} a_{ij} x^i y^j = \sum_{j=0}^{m} \left(\sum_{i=0}^{n} a_{ij} x^i \right) y^j.$$

Its value can be obtained by successively computing the values of the $m + 2$ one-dimensional polynomials

$$b_j := b_j(x) := \sum_{i=0}^{n} a_{ij} x^i, \quad j = 0(1)m, \tag{9.7.14}$$

and

$$p(x, y) := \sum_{j=0}^{m} b_j y^j. \tag{9.7.15}$$

The results of the computation (9.7.14) are long intervals. The final result in (9.7.15) is also a long interval. It may be rounded into a floating-point interval if desired.

Long interval arithmetic has also been very successfully applied to the computation of orbits of discrete dynamic systems. It is well known that such computations are highly unstable if the system exhibits chaotic behavior. In this case even for very simple systems ordinary floating-point arithmetic delivers results which are completely wrong. Also ordinary interval arithmetic (i.e., intervals of floating-point numbers)