SIAM REVIEW
Vol. 28, No. 1, March 1986

THE ARITHMETIC OF THE DIGITAL COMPUTER: A NEW APPROACH*

U. W. KULISCH[†] AND W. L. MIRANKER[†]

Dedicated to the Memory of Phyllis Miranker

Abstract. A new approach to the arithmetic of the digital computer is surveyed. The methodology for defining and implementing floating-point arithmetic is described. Shortcomings of elementary floating-point arithmetic are revealed through sample problems. The development of automatic computation with emphasis on the user control of errors is reviewed. The limitations of conventional rule-of-thumb procedures for error control in scientific computation are demonstrated by means of examples. Computer arithmetic is extended so that the arithmetic operations in the linear spaces and their interval correspondents which are most commonly used in computation can be performed with maximum accuracy on digital computers. A new fundamental computer operation, the scalar product, is introduced to develop this advanced computer arithmetic.

A process of automatic error control called validation which delivers high accuracy with guarantees for scientific computations is described. Validation of computations for a large class of numerical problems is made possible by advanced computer arithmetic. High accuracy is furnished by coupling the scalar product with the process of defect correction. Guarantees and error bounds are obtained by interval techniques. This whole process establishes certain numerical algorithms such as the evaluation of rational expressions as additional higher order arithmetic operations. The development of some programming languages in the context of computer arithmetic is reviewed. A collection of constructs in terms of which a source language may accommodate the methodology of computer arithmetic in a user-friendly mode is described. Finally the current state of implementation of the ideas discussed here is reviewed.

Key words. floating-point, fully accurate inner product, semimorphism, computation with guarantees

AMS(MOS) subject classifications. 65G05, 65G10, 65G99

CONTENTS

1. Introduction.

1. Introduction. Historically, computers were developed for scientific computation. Today, the digital computer is a general purpose machine. It is used in such diverse areas as game playing, banking, reservation systems, traffic control, language

*Received by the editors July 30, 1984, and in revised form July 1, 1985.

[†] University of Karlsruhe, Karlsruhe, West Germany and IBM T. J. Watson Research Center, Yorktown Heights, New York 10598. This article was partially prepared during a stay of the authors at the Computer Science Department of the University of California at Santa Barbara, in February–March, 1984.

translation and inventory control. Because of this proliferation of computer usage, it is easy to overlook the central relationship between the computer and scientific computation.

There are two principal number systems used in a modern digital computer. These are integer systems (fixed-point systems) and floating-point systems. These number systems require different concepts of computer arithmetic. The integer system is to a large extent the system used in the area of nonscientific computation of the types enumerated above. As long as the values of computed results do not exceed the range of representable integers (i.e., as long as no overflow and no noninteger result occurs), these computations are error-free. For this reason, the public image of the computer is one of a perfect computational tool.

Problems of scientific computation occur everywhere in the natural sciences and in technology. Examples of such problems are solving a differential equation or a system of algebraic equations. The floating-point system along with the operations of floating-point arithmetic are used as an approximate means for calculating solutions of such problems. Floating-point arithmetic confronts us with a seemingly paradoxical situation. On one hand, many modern computers perform the basic floating-point operations with high, even maximum accuracy. Nevertheless, the results of a scientific computation composed of several of these operations may be grossly incorrect. As an example of this consider the determination of the following sum.

$$10^{50}+812-10^{50}+10^{35}+511-10^{35}=1323.$$

By summing these numbers from left to right, most digital computers will return 0 (zero) as the answer. This error comes about because the floating-point formats in these computers are unable to cope with the large digit range required for this calculation. The obvious solution for this particular example is to exchange the operands in an appropriate way. Such problem fixes are not always known. Even when they are known, they cannot always be applied for practical reasons. We shall give several additional examples of the failure of computers to deliver correct results later on.

This article deals with floating-point arithmetic from a contemporary point of view. We shall show that recently developed concepts and methods of floating-point arithmetic provide a superior capability for modern digital computers with far-reaching consequences for scientific computation. For example, they go a long way toward eliminating errors of the type just described. There are other nonfloating-point arithmetic implementations for eliminating error in scientific computation. Examples of these are rational arithmetic, the use of multiple precisions and the full precision arithmetic found in such systems as SCRATCHPAD and MACSYMA. We stress that our methodology is to enhance the practical high performance quality of floating-point with the safety which is provided with these other methods.

We begin this development in §2 with a description of floating-point numbers and elementary floating-point arithmetic. The methodology for defining and implementing floating-point arithmetic is informally described. Shortcomings of elementary floating-point arithmetic are revealed through sample problems.

In §3 we give a brief review of the development of automatic computation with emphasis on the user control of errors. Rule-of-thumb procedures for error control employed in scientific computation are discussed. Limitations of these procedures are demonstrated by means of examples. This motivates the necessity for the further development of computer arithmetic which follows.

In §4 we extend computer arithmetic so that the arithmetic operations in the linear spaces and their interval correspondents which are most commonly used in computation can be performed with maximal accuracy on digitial computers. A new fundamental computer operation, the scalar product, is introduced to develop this advanced computer arithmetic.

In §5 a process of automatic error control called validation is described. Validation delivers high accuracy with guarantees for scientific computations. Validation of computations for a large class of numerical problems is made possible by advanced computer arithmetic. High accuracy is furnished by coupling the scalar product with a special numerical process called defect correction. Guarantees and error bounds are obtained by interval techniques. This whole process establishes certain numerical algorithms such as the evaluation of rational expressions as additional higher order arithmetic operations.

In §6 we review the development of some programming languages in the context of computer arithmetic. We describe a collection of constructs in terms of which a source language (such as FORTRAN or PASCAL) may accommodate the methodology of computer arithmetic in a user-friendly mode. For this, we organize computer arithmetic into three levels of implementation. The first level, called basic arithmetic, deals with elementary computer arithmetic augmented by the scalar product. The second level is concerned with advanced computer arithmetic and its setting in linear spaces of computation. The third level treats the validation process, including the capability of a source language to conveniently express the evaluation of expressions such as rational functions with maximum accuracy.

In §7 the current state of implementation is reviewed.

Many of the computational examples used in this article, have been taken from well-known collections [12], [25]. We stress that this is not a review paper on computer arithmetic. It is a survey of the new approach to this subject which has been developed by ourselves and a number of collaborators in recent years. For this reason the large body of work which deals with computer arithmetic but which has not directly contributed to this new approach is neither surveyed nor referred to. For convenience to the reader we do include a supplementary bibliography of important work in computer arithmetic outside of the new approach.

We view as a high point of the new approach a coherency with which it addresses the subject matter. A simple but rigorous mathematical foundation for computer arithmetic is given. Applications and reduction to practice in scientific computation of these ideas is included. Finally, implementations in hardware and software are also described. We believe that new prospects for computation are likely as a result [16].

2. Floating-point numbers and elementary floating-point arithmetic. The real numbers can be defined axiomatically as a conditionally complete linearly ordered field $\mathbb{R}$. Independently of what this this abstract idea means, we are familiar with decimal expansions in terms of which real numbers may be represented. Decimal expansions employ a base $b=10$. In the case of a general base, such an expansion has the following form:

$$(1) \qquad x = * \, d_n d_{n-1} \cdots d_1 d_0 . d_{-1} d_{-2} d_{-3} \cdots = * \sum_{i=n}^{-\infty} d_i b^i,$$

where $* \in \{+,-\}$ and b is an integer greater than unity. The d_i, $i=n(-1)-\infty$, are integers between zero and $b-1$. That is,

$$(2) \qquad 0 \leqq d_i \leqq b-1 \quad \text{for all } i=n(-1)-\infty.$$

For technical reasons stemming from the requirements of the uniqueness of representations of the form (1), we also require that

$$d_i \leqq b-2 \quad \text{for infinitely many } i. \tag{3}$$

In (1) b is called the base or the radix of the number system. The point between d_0 and d_{-1} is called the radix point, i.e., the decimal point when $b=10$. The d_i, $i=n(-1)-\infty$, are called the digits (of base b). (When $b=2$ the digits are called bits.)

Arithmetic operations for these infinite b-expansions are defined by means of successive approximations. Let x and y be two real numbers. Truncation of the b-expansion of x and y after the rth digit after the radix point gives the truncated expansions x_r and y_r, respectively. For any of the arithmetic operations $* \in \{+,-,\times,/\}$, the result $x_r * y_r$ can be calculated following well-known rules. The operation $x*y$ for $* \in \{+,-,\times,/\}$ for the full b-expansions is then defined as the limit of the sequence $x_r * y_r$, obtained by letting r go to infinity. Such a limiting process cannot be executed in a finite time. Thus for an approximation of the real numbers and operations, floating-point numbers and floating-point operations are used. Such numbers are representable and such operations are implementable on a computer.

A normalized floating-point number x (in sign-magnitude representation) is a real number x in the form

$$x = * \, m b^e.$$

Here $* \in \{+,-\}$ is the sign of the number (sign(x)), m is the mantissa (mant(x)), b is the base of the number system in use and e is the exponent (exp(x)); b is an integer greater than unity. The exponent is an integer between two fixed integer bounds $e1$, $e2$, and in general, $e1 \leqq 0 \leqq e2$. The mantissa m is of the form

$$m = \sum_{i=1}^{l} d[i] b^{-i}.$$

The $d[i]$ are the digits of the mantissa. They have the properties $d[i] \in \{0,1,\cdots,b-1\}$ for all $i=1(1)l$ and $d[1] \neq 0$. Without the condition, $d[1] \neq 0$, floating-point numbers are said to be not normalized. The set of normalized floating-point numbers does not contain zero. For a unique representation of zero we assume that sign(0) $= +$, mant(0) $=0.00\cdots0$ (l zeros after the radix point) and exp(0) $= e1$. A floating-point system depends on the constants $b, l, e1$, and $e2$. We denote it by $R = R(b,l,e1,e2)$.

A floating-point system R consists of a finite number of elements. They are equally spaced between successive powers of b and their negatives. This spacing changes at every power of b. Figure 1 shows a simple floating-point system $R = R(2,3,-1,2)$ consisting of 33 elements [12]. The successive powers of 2 are $\pm\frac{1}{4}$, $\pm\frac{1}{2}$, ± 1, ± 2. The floating-point system R has a greatest and a least element. Each number in R has to represent an entire interval of real numbers. For instance, in Fig. 1 the floating-point number 3 might represent the indicated shaded interval. A floating-point system has the appearance of a screen placed over the real numbers. Indeed, the expression floating-point screen is often used.

Next we turn to the arithmetic operations $+,-,\times,/$. These operations for real numbers are approximated by floating-point operations. If x and y are floating-point numbers, the exact point $x \times y$ itself is not usually a floating-point number of $R(b,l,e1,e2)$ since the mantissa of $x \times y$ has $2l$ digits. For related reasons, the exact sum $x+y$ is also not usually a floating-point number. Since a computer must be able to

e	2^e	m_1	m_2	m_3	m_4
-1	$1/2$	0.100	0.101	0.110	0.111
0	1	0.100	0.101	0.110	0.111
1	2	0.100	0.101	0.110	0.111
2	4	0.100	0.101	0.110	0.111

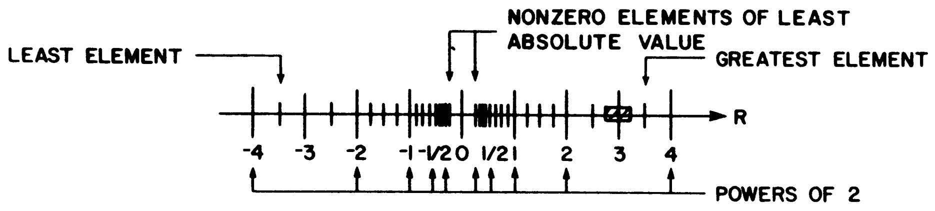


FIG. 1. *A simple floating-point system.*

represent the results of its own operations, the result of a floating-point operation must be a floating-point number. The best we can do is to round the exact result into the floating-point screen and take the rounded version as the definition of the floating-point operation.

If $*$ is one of the exact operations, $+, -, \times, /$, let $\boxed{*}$ denote the corresponding floating-point operation. Then our choice of floating-point operations is expressed by the following mathematical formula.

(RG) $\quad x \boxed{*} y := \square(x * y) \quad$ for all $x, y \in R$ and all $* \in \{+, -, \times, /\}$.

In (RG), $\square$ is a mapping $\square$: $\mathbb{R} \to R$. $\square$ is called a rounding if it has the following properties (R1) and (R2).

(R1) $\qquad \square x = x \quad$ for all $x \in R$,

that is, the screen R is invariant under the mapping $\square$.

(R2) $\qquad x \leqq y \Rightarrow \square x \leqq \square y \quad$ for all $x, y \in \mathbb{R}$,

that is, $\square$ is monotonic on the real numbers.

The three familiar roundings: to the nearest floating-point number, toward zero or away from zero have properties (R1) and (R2) and the following additional property.

(R4) $\qquad \square(-x) = -\square x \quad$ for all $x \in \mathbb{R}$.

We impose this requirement of antisymmetry on many roundings.

Later on we shall develop arithmetic techniques for supplying guarantees in floating-point computation. For these techniques, we need the monotone upwardly and the monotone downwardly directed roundings $\triangle$ and ∇. These two roundings are characterized by (R1), (R2) and the additional property

(R3) $\qquad \nabla x \leqq x$ and $x \leqq \triangle x \quad$ for all $x \in \mathbb{R}$.

Thus, ∇ rounds to the left and $\triangle$ rounds to the right. However, the roundings ∇ and $\triangle$ do not have the antisymmetry property (R4).

All operations defined by (RG) and a rounding with the properties (R1)–(R3) produce results of maximum accuracy in a certain sense which is rounding dependent.

In particular, between the correct result (in the sense of real numbers) and the approximate result $x \boxed{*} y$ (in the sense of the screen of floating-point numbers) no other floating-point number in the screen can be found.[1]

The proof of this property follows easily from (RG), (R1) and (R2).

Proof. Assume that u, $v \in R$ are two adjacent floating-point numbers with the property $u \leqq x * y \leqq v$. Then from (R2) we obtain $\square u \leqq \square(x * y) \leqq \square v$. Then (R1) and (RG) deliver the desired result $u \leqq x \boxed{*} y \leqq v$. $\square$

For convenience, we shall refer to the class of roundings which satisfy (R1), (R2), and (R4) along with the special roundings $\triangle$ and ∇ as admissible roundings. We may summarize this discussion by saying that admissible roundings generate maximally accurate floating-point arithmetic through use of (RG).

Algorithms for implementation of the operations defined by (RG) and admissible roundings which are used on many computers can be found in the literature [15], [18], [19], [21]. Here we review the main features of implementation.

At first sight it seems to be doubtful that formula (RG) can be implemented on computers at all. In order to determine the approximation $x \boxed{*} y$, the exact but unknown result $x * y$ which is in general neither computer specifiable nor computer representable seems to be required in (RG). It can be shown, however, that whenever $x * y$ is not representable on the computer, it is sufficientto replace it by an appropriate and representable value $x \,\tilde{*}\, y$. The latter has the property $\square(x * y) = \square(x \,\tilde{*}\, y)$ for all roundings in question. Then $x \,\tilde{*}\, y$ can be used to define $x \boxed{*} y$ by means of the relations

$$x \boxed{*} y = \square(x * y) = \square(x \,\tilde{*}\, y) \quad \text{for all } x,\ y \in R.$$

There are fast algorithms for an implementation of (RG) on computers. These algorithms consist of the following five steps:

1. Decomposition of x and y, i.e., separation of x and y into mantissa and exponent. If a floating-point number is not stored in a single word, this step is vacuous.
2. Determination of $x \,\tilde{*}\, y$. It may be that $x \,\tilde{*}\, y = x * y$.
3. Normalization of $x \,\tilde{*}\, y$. $x \,\tilde{*}\, y$ requires normalization if its mantissa has one or more zero digits following the radix point. Normalization consists of repeatedly shifting the mantissa left by one digit and decreasing the exponent by unity until all such zeros are eliminated. A single shift right may also be necessary in the case of addition. If the result of 2 is already normalized, this step can be skipped.
4. Rounding of $x \,\tilde{*}\, y$ determines $x \boxed{*} y = \square(x \,\tilde{*}\, y) = \square(x * y)$.
5. Composition, i.e., assembling of the mantissa and exponent of the result into a floating-point number. If floating-point numbers are not stored in single words, this step is vacuous.

Figure 2 shows a graphical representation of these five steps in the form of a flow diagram. Since we deal with monotone roundings only, the normalization has to be performed before the rounding, since otherwise the monotonicity of the rounding is lost. Division can be executed in a manner that eliminates the need for normalization.

[1]We shall introduce the term maximal-$\square$ accuracy later to describe this concept of accuracy for a class of computer operations, since the accuracy depends on the rounding $\square$. For convenience we drop the suffix (-$\square$), since confusion will not occur.

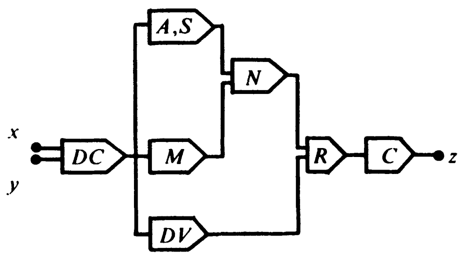


FIG. 2. *Flow diagram for the arithmetic operations DC: decomposition; A, S: addition and subtraction; M: multiplication; DV: division; N: normalization; R: rounding; C: composition.*

In the implementation of (RG) it is essential that $x \boxed{*} y$ is produced by $\square(x * y)$ *for all* $x, y \in R$. This can only be achieved if the accumulator that performs the operations is long enough. There are still many computers in the marketplace which for the execution of the floating-point operations use an accumulator which is only as long as the floating-point mantissa. We shall presently use a simple example to show that (RG) cannot be strictly realized with such an accumulator. While there are many tricky ways to implement floating-point arithmetic, there have emerged two standard approaches to this implementation which we shall discuss: the implementations by a so-called long accumulator and by a so-called short accumulator. These two accumulators accommodate all admissible roundings of interest. The long accumulator is a computer register with one digit, which may be a binary digit,in front of the radix point and $2l+1$ digits of base b after the radix point. See Fig. 3a. The short accumulator is a computer register with one digit, which can be a binary digit, in front of the radix point and $l+2$ digits of base b plus one binary digit after the radix point. See Fig. 3b.

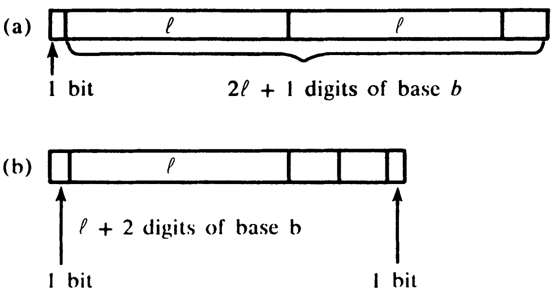
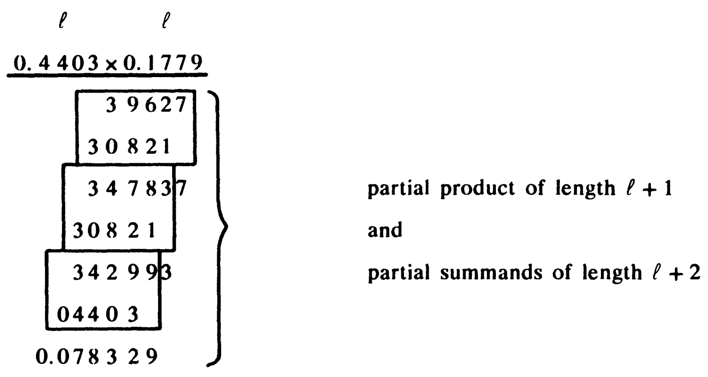


FIG. 3. (a) *Long accumulator*; (b) *short accumulator.*

An accumulator shorter than the short accumulator cannot always deliver correct and optimal results (in the sense which we have specified) for the floating-point operations. The bit on the left end of both accumulators is used for a possible overflow which may occur in case of addition. If the short accumulator is used for multiplication, the mantissa of the product has to be built up from the right as illustrated by the following illustration. The bit on the right end of the short accumulator is needed in the cases of the roundings $\triangle$ and ∇.


ℓ ℓ
0.4403 × 0.1779
39627
30821
347837
30821
342993
04403
0.078329
partial product of length ℓ + 1
and
partial summands of length ℓ + 2

$0.4403 \boxtimes 0.1779 = 0.7833 \times 10^{-1}$

We now give a simple example which shows that any reduction of the length of the accumulator causes a failure to deliver the optimal results we have specified. Take $l = 4$ and the decimal system $b = 10$. We show that an accumulator of $l + 1 = 5$ digits followed by an additional binary digit d after the point is not capable of delivering correct results as defined by (RG) in all cases. Let $x = 0.1000 \times 10^6$, $y = -0.5001 \times 10^1$, so that $x + y = 0.099994999 \times 10^6$. If we now apply the rounding to the nearest floating-point number (to four decimal digits), we obtain: $x \boxplus y = 0.9999 \times 10^5$. However execution in an accumulator of 5 decimal digits leads to a different result, namely 0.1000×10^6.

In practice, the choice between the short and long accumulators depends on side considerations such as the technology employed and fine points in the design. However, a fundamental perception of numerical analysis is that advanced optimal methods of computer arithmetic (which we shall develop below) require the accumulation of the full double length product of two floating-point numbers. Such double length products cannot be efficiently produced by the short accumulator.This consideration gives very high priority to the choice of the long accumulator for the execution of floating-point operations. Indeed use of the short accumulator would require a complicated simulation process for accommodating the double length products needed for the approach to high accuracy computer arithmetic in product spaces which we develop below.

Although this requirement for the double length product is well known in numerical computation, many processors continue to be built without this feature, some even adhering to a currently fashionable (albeit dubious) claim of furnishing high accuracy in computation.

Let us return to our earlier point that although floating-point operations with maximum accuracy can be implemented and realized in computers, results of scientific computations composed of these operations may be grossly incorrect. All mathematical statements depend critically on the premises upon which they are built. Arithmetic expressions or numerical algorithms are not exempted from this requirement. If compromises are made such as the replacement of full precision addition or the replacement of the full set of real numbers by a finite set of floating-point numbers, we are obliged to accept compromises in the result of evaluating that expression or executing that numerical algorithm. Perhaps what is surprising is that the discrepancies in the results can be catastrophically large even though the compromises in the premises are quite small. We illustrate this phenomenon with a few examples.

1. Cancellation. Consider a floating-point system with the base $b=10$ and a mantissa of 5 digits. We compute $x \boxminus y$, where

$$x=0.10005\times10^5 \quad\text{and}\quad y=-0.99973\times10^4.$$

Using the short accumulator, we get

$$\begin{array}{rr} 0.10005\times10^5-0.99973\times10^4= & 0.1000500\times10^5 \\ & \underline{-0.0999730\times10^5} \\ & 0.0000770\times10^5 \\ \text{normalized:} & 0.7700000\times10^1 \\ \text{rounded:} & 0.77000\quad\times10^1 \end{array}$$

That is, $x\boxminus y=0.77000\times10^1$. The occurrence of leading zero digits after the decimal point is called cancellation. The process of normalization then fills in zeros at the right end of the result. The rounding has no effect, i.e., the result is error free even in the sense of an exact subtraction of real numbers.

Now suppose that each of the floating-point numbers x and y are themselves rounded results of products of two floating-point numbers, i.e., $x=\square(x_1\times x_2)$ and $y=\square(y_1\times y_2)$. The products $x_1\times x_2$ and $y_1\times y_2$ which have mantissas of double length are taken to be

$$x_1\times x_2=0.10005482410\times10^5,$$

$$y_1\times y_2=0.09997342213\times10^5.$$

Rounding gives the values of x and y used previously. Subtracting, we now get

$$\begin{array}{rr} x_1\times x_2-y_1\times y_2= & 0.10005482410\times10^5 \\ & \underline{-0.09997342213\times10^5} \\ & 0.00008140197\times10^5 \\ \text{normalized:} & 0.8140197\times10^1 \end{array}$$

That is, $x_1\times x_2-y_1\times y_2=0.8140197\times10^1$.

Comparison with the result $x\boxminus y=x_1\boxtimes x_2\boxminus y_1\boxtimes y_2$ obtained earlier shows that no digits of the mantissas coincide. The results agree only in magnitude. Cancellation occurs whenever two nearly equal numbers are subtracted. Although the single subtraction step is error free, cancellation is very dangerous if the data themselves are already rounded.

Cancellation is the cause for many failures in floating-point computation. Note that the result $\square(x_1\times x_2-y_1\times y_2)$ prescribed by our methods (compare (RG) above) can be obtained if the products $x_1\times x_2$ and $y_1\times y_2$ are computed to their full double length and then subtracted using the long accumulator. Indeed, in this case we get

$$\square(x_1\times x_2-y_1\times y_2)=0.81402\times10^1,$$

an optimal result since no floating-point number lies between it and the exact result. Cancellation may also occur over a long chain of additions/subtractions. In this case, one speaks of global or catastrophic cancellation. Long computations tend to conceal the occurrence of global cancellation as the following example shows.

2. Global cancellation. Using Taylor series, the following formula for the exponential is derived.

$$e^z=\sum_{v=0}^{\infty}\frac{z^v}{v!}.$$

This series is absolutely convergent for every value of z in the complex plane. For z real and negative the terms in the series alternate in sign. In this case the magnitude of the error committed by truncating such an alternating series is less than the magnitude of the first term neglected. Let us use this series to calculate the value of the exponential for $z=-20$ and employing a floating-point system with 6 decimal digits in the mantissa. In Fig. 4, we display a list of the terms of the series for $v=1(1)62$. The

```
x = -2.00000000000000E+01

 0             1.00000000000000000
 1           -20.00000000000000000
 2           200.00000000000000000
 3         -1333.33000000000000000
 4          6666.65000000000000000
 5        -26666.60000000000000000
 6         88888.70000000000000000
 7       -253968.00000000000000000
 8        634920.00000000000000000
 9      -1410930.00000000000000000
10       2821860.00000000000000000
11      -5130650.00000000000000000
12       8551080.00000000000000000
13     -13155500.00000000000000000
14      18793600.00000000000000000
15     -25058100.00000000000000000
16      31322600.00000000000000000
17     -36850100.00000000000000000
18      40944600.00000000000000000
19     -43099600.00000000000000000
20      43099600.00000000000000000
21     -41047200.00000000000000000
22      37315600.00000000000000000
23     -32448300.00000000000000000
24      27040300.00000000000000000
25     -21632200.00000000000000000
26      16640200.00000000000000000
27     -12326100.00000000000000000
28       8804360.00000000000000000
29      -6071970.00000000000000000
30       4047980.00000000000000000
31      -2611600.00000000000000000
32       1632250.00000000000000000
33       -989242.00000000000000000
34        581907.00000000000000000
35       -332518.00000000000000000
36        184732.00000000000000000
37        -99855.10000000000000000
38         52555.30000000000000000
39        -26951.40000000000000000
40         13475.70000000000000000
41         -6573.51000000000000000
42          3130.24000000000000000
43         -1455.93000000000000000
44           661.78600000000000000
45          -294.12700000000000000
46           127.88100000000000000
47           -54.41740000000000000
48            22.67390000000000000
49            -9.25465000000000000
50             3.70186000000000000
51            -1.45171000000000000
52             0.55835000000000000
53            -0.21069800000000000
54             0.07803630000000000
55            -0.02837680000000000
56             0.01013460000000000
57            -0.00355600000000000
58             0.00122621000000000
59            -0.00041566400000000
60             0.00013855500000000
61            -0.00004542790000000
62             0.00001465420000000

     ..................................

     ----------------------------------
             181.49600000000000000

               0.00000000206115362
```

FIG. 4. *Floating-point summation of series for exponential.*

computed value of the sum of these terms is 181.496. The summation was stopped at this point because the last summand is less than 10^{-7} times the sum to the indicated terms. Stopping at this point is conventional numerical practice, since according to the error property of the alternating series already noted, further summing ought not to influence the computed result. However, the correct result is 0.00000000206115.... We have drawn a vertical line in the display of the summands between the 8th and 9th places after the decimal point. As we now see from the correct results, all digits of every summand to the left of this line should cancel.

The correct answer is of the order 10^{-9} while the computed floating-point result is of the order 10^3. More cannot be expected of a result computed with 6 decimal places. To see this note that the largest summand corresponds to $v=20$, and that it has the value 43099600. This summand cannot be correct to more than 6 places. Thus, the first two places to the left of the decimal point of this summand have no meaning, and any sum involving them can likewise have no meaning in these two places. Then the global cancellation to the left to the indicated vertical line can likewise not occur in these two places except by the sheerest accident. In fact, the required cancellation does not occur in the computation, and so, the leading digits of the computed sum indicated are incorrect. The reader should try to compute the solution of the following examples (taken from [25]) with his pocket calculator, personal computer or by use of a mainframe by himself. The correct result for each of the problems is given, and in most cases, the answer obtained by a computer using a floating-point system with a 14 hexadecimal digit mantissa (i.e., base 16 or approximately 17 decimal digits) is also given.

3. Scalar products. Calculate the scalar product of two vectors A and B with five elements each:

$$SP=A1\times B1+A2\times B2+A3\times B3+A4\times B4+A5\times B5$$

for

$$\begin{aligned} A1 &= 2.718281828, & B1 &= 1486.2497,\\ A2 &= -3.141592654, & B2 &= 878366.9879,\\ A3 &= 1.414213562, & B3 &= -22.37492,\\ A4 &= 0.5772156649, & B4 &= 4773714.647,\\ A5 &= 0.3010299957, & B5 &= 0.000185049. \end{aligned}$$

The correct value of the scalar product is

$$-1.00657107\times 10^{-11}.$$

The computer delivers

$$+0.335\ldots\times 10^{-9},$$

so that even the sign is incorrect. Note that no vector element has more than 10 decimal digits.

4. Arithmetic expressions. Evaluate the arithmetic expression

$$(1682XY^4+3X^3+29XY^2-2X^5+832)/107751$$

for

$$X=192119201 \quad\text{and}\quad Y=35675640.$$

The correct answer is 1783. The computer delivers

$$-5.385\ldots\times 10^{22}.$$

5. Polynomial evaluation. Evaluate the polynomial

$$P(X)=8118X^4-11482X^3+X^2+5741X-2030$$

for

$$X=0.707107.$$

The correct value of the polynomial is

$$-1.91527325270\ldots\times 10^{-11}.$$

The computer delivers

$$P(X)=-1.97815097635611891\times 10^{-11}.$$

6. Linear equations. Solve the set of equations

$$64919121X-159018721Y=1,$$
$$41869520.5X-102558961Y=0.$$

Expressions to evaluate X and Y exactly are

$$Y=(41869520.5/64919121)/(102558961-41869520.5\times 159018721/6491912),$$
$$X=(102558961/41869520.5)Y.$$

The correct results are

$$X=205117922, \qquad Y=83739041.$$

The computer delivers

$$X=0.987372352669808606\times 10^{-1}, \qquad Y=0.403093099594116210\times 10^{-1}.$$

What result does your computer deliver?

7. Extrapolation. The following values are given

X	5201477	5201478	5201479
Y	99999	100000	100001

Obviously the three values fit on a line. Therefore, a best linear approximation $L(x)=mx+b$ must yield $L(5201480)=100002$. Formulas for the computation of m and b are

$$m=\frac{X1\times Y1+X2\times Y2+X3\times Y3-\frac{1}{3}(X1+X2+X3)(Y1+Y2+Y3)}{X1+X2^2+X3^2-\frac{1}{3}(X1+X2+X3)^2},$$

$$b=\frac{1}{3}(Y1+Y2+Y3)-\frac{m}{3}(X1+X2+X3).$$

Evaluate m and b using these formulas and determine $L(5201480)$. The correct results are $m=1$, $b=-510478$ and $L(5201480)=100002$.

8. Differentiation. Consider the function

$$f(t)=\frac{4970t-4923}{4970t^2-9799t+4830}.$$

An approximation for the value of the second derivative $f''(t)$ of a function $f(t)$ may be computed from the expression

$$\frac{f(t-h)-2f(t)+f(t+h)}{h^2}$$

with some small value for h. Determine an approximation of $f''(1)$ with the help of the above expression for $h=10^{-4}$, $h=10^{-5}$, $h=10^{-8}$. The correct results are:

Approximation with $h=10^{-4}$: 70.78819....

Approximation with $h=10^{-5}$: 93.76790....

Approximation with $h=10^{-8}$: 94.00000....

The exact value for the derivative is $f''(1)=94$. The computer delivers:

Approximation with $h=10^{-4}$: 70.7804197738837856....

Approximation with $h=10^{-5}$: 93.1278568018018011654....

Approximation with $h=10^{-8}$: 30695.4411053317471....

9. Expression evaluation. Compute the value of the expression

$$83521y^8+578x^2y^4-2x^4+2x^6-x^8$$

for $x=9478657$ and $y=2298912$. The correct value is

$$-179689877047297.0.$$

What is the result obtained on a pocket calculator? on a large computer?

10. Complex division. Compute the quotient of two complex numbers

$$(a+ib)/(x+iy)$$

for

$$a=1254027132096, \qquad x=886731088897,$$
$$b=886731088897, \qquad y=627013566048.$$

The correct value of the quotient is

$$1.41421\ldots+i8.47861\ldots\times 10^{25}.$$

These examples show that computers supplied with the best possible implementation of the four arithmetic operations $+,-,\times,/$ can deliver arbitrarily bad results in problems of the simplest form. One may imagine the possible implications of incorrect results in computation for more serious purposes such as power grids, reactor management, weapon systems, aircraft design and control, vehicle stability and so on.

The reader should not be discouraged if his or her attempts to solve these problems did not produce the correct answer. Most computers in the marketplace today can do no better. The examples make a bad case for floating-point arithmetic. How is it that the digital computer has for many years been used very successfully in numerical computation? Numerical analysts have developed great skills and sophisticated methods to detect such errors and to maneuver around them. These specialized techniques require extensive study and much experience to be used. Moreover their use adds considerable time and expense to the computational process. Not every computer user is sophisticated and experienced. The inexpert user is often unable to detect such errors and is usually at a loss of how to proceed when such errors occur.

In the following sections we show how floating-point arithmetic has been advanced so that the digital computer can automatically control and rectify many errors inherent in floating-point computation. Indeed, a new capability called validation is possible in many cases, whereby the computer gives a result and an absolute assertion of its accuracy. In this manner, the computer becomes a precise scientific instrument rather than an experimental tool.

3. Historic remarks and motivation. The Roman number system is hardly one upon which to build a computing machine. What is surprising is that this ancient system was in widespread use in Europe up to the 15th century. It was supplanted by the Arabic number system, itself a system dating from antiquity, in other parts of the world. Once the Arabic number system became widely adopted in Western Europe, mechanical calculating devices of all sorts began to appear. Better known examples of these devices are associated with the names of Blaise Pascal and Gottfried Leibniz. Pascal is credited with having built and used an adding machine. A little later Leibniz invented the principle of the stepping cylinder (Staffelwalze) by means of which it was possible to perform all four operations of arithmetic directly. This device, in one form or another, could be found in mechanical calculators up to the present day.

The realization that the intellectual process of computation could be implemented by mechanical devices was a major fundamental discovery. This discovery created an industry which developed further principles and concepts of mechanical computation, as well as devices by means of which they were implemented. The descendants of these venerable firms could be found pursuing the same enterprise well into the 20th century. Some of them are currently in the electronic computer business.

With the invention of the logarithm by John Napier and others, the appearance of the slide rule soon followed. This was an essential step in the development of analog computing devices. Although not as widely spread as the digital computer, analog devices still exist today in sophisticated electronic and mechanical form.

The relatively slow speed of mechanical computing devices supported an interactive mode of computation whereby the user monitored the result of each operation as it was produced. Thus, error control and significance of results could be dealt with by the user's understanding of what was going on. Many of us who are familiar with the use of a slide rule or a product calculator have performed this kind of error control of a machine aided computation. A rule of thumb had it that in this mode of interactive computation, a person could perform about 1000 reliable computations per day. This translates into approximately 0.03 operations per second for a nine hour day.

In the period 1920–1940, a breakthrough in computation was made. This was the idea of the stored program computer in which the program itself could be stored in the computer and operated upon by the computer as if it were data. This breakthrough is credited variously to Alan Turing, Emil Post, John Mauchly and John von Neumann. Combining this with the technological electronic developments of the 20th century led to the first generation of modern digital computers. These computers provided a gigantic gain in computer power over their mechanical predecessors. In the early fifties, these computers were able to execute on the order of 30 floating-point operations per second which, in fact, were implemented as subroutine calls. This was a thousandfold gain in speed. The modern computer age is dated from this period.

Early electronic computers often represented their data as fixed-point numbers. This imposed a scaling requirement. Problems had to be pre-processed by the user so that they could be accommodated by this fixed-point number representation. This pre-processing proved to be an enormous burden. It was the introduction of the floating-point representation in computation in the early fifties which largely eliminated this burden. But it turned out that the floating-point representation made the error control problem even more difficult. There was no longer any hope for error control of computations by the traditional interactive methods used in the cases of mechanical desk calculators or the slide rule.

The enormous gain in speed and the introduction of floating-point mandated the development of methods for more systematic control of errors in computation. Such methods, that were developed in those days and are still used today, are based on estimates of the error of each individual arithmetic operation. These ideas and concepts trace back to Cornelius Lanczos and Wallace Givens and were heavily exploited by James H. Wilkinson and others. These methods are highly sophisticated. They led to the two techniques of error analysis commonly called forward and backward error analysis. Both are analytic methods. Since the computer is able to execute a large number of operations, a large number of error estimates have to be made and their propagation through the whole algorithm has to be studied. For instance, multiplication of two complex matrices of 100 rows and columns requires about 8 million such estimates. The propagation of these estimates in a complicated algorithm requires a rather complicated analysis which can only be performed in special types of problems. Even then, the results are usually theoretical and of limited practical value. Indeed, even sophisticated users tend to avoid this approach.

Thus, we find that other methods for judging the quality of results delivered by a computation have arisen. We find many computers equipped with both single and double precision and sometimes even with extended capabilities of precision in arithmetic. The scientific computer user usually adopts one of the following techniques for judging the quality of his output.

1. He computes a residual, i.e., he inserts the computed answer into the problem expression and evaluates the remainder, *hoping* that a small remainder indicates a good solution.
2. He repeats his calculation in double or extended precison, checking for agreement, *hoping* that good agreement indicates a good solution.
3. He reruns his problem with slightly changed input data, checking the variation in the results. Small variation is interpreted as stability in the computational process and *hopefully* a good solution.

These approaches frequently give good indication of the quality of a computation. However, they may also be completely unreliable.

The following pair of equations [12] shows how unreliable method 1 may be.

$$0.780X+0.563Y=0.217,$$
$$0.913X+0.659Y=0.254.$$

Two different approximate solutions are proposed.

$$\begin{array}{lll} & X=0.999, & Y=-1.001, \\ \text{and} & X=0.341, & Y=-0.087. \end{array}$$

Which one is better? The usual check is to substitute them into the set of linear equations. We find the following residuals:

$$\begin{array}{ll} & 0.780X+0.563Y-0.217=-0.001243, \\ & 0.913X+0.659Y-0.254=-0.001572, \\ \text{and} & \\ & 0.780X+0.563Y-0.217=-0.000001, \\ & 0.913X+0.659Y-0.254=0. \end{array}$$

It seems evident that the second approximation is a better solution, since it makes the residuals much smaller, $(-0.000001, 0.)$ compared to $(-0.001243, -0.001572)$. However, the true solution is $X=1$, $Y=-1$, as one can verify easily. Hence, the first approximation ($X=0.999$, $Y=-1.001$) is much closer to the true solution.

The second traditional approach for checking the accuracy of a computed result is to recalculate, each time increasing the number of digits with which computations are performed. Thus, we find most computers equipped with single and double precision and sometime even extended precison capabilities in arithmetic. The idea underlying this second approach is closely related to the definition of the operations for the real numbers through limiting processes which we discussed previously.

The result of such an operation was defined as being the limit of the result obtained by operating on truncated parts of the expansions representing the operands. However, the analogy is only superficial. Indeed this approach only displaces the problem, but does not solve it in principle. It is evident that the examples displayed above have counterparts which demonstrate equivalent deficiencies in the double or extended precision computation of any computer. For our simple example,

$$10^{50}+812-10^{50}+10^{55}+511-10^{55}=1323,$$

almost all digital computers will return zero, whether using single, double or extended precision. In general, the user does not know how many digits are needed to obtain a correct answer. To show that the third method is also unreliable, consider the two linear equations

$$100000x+99999y=b_1,$$
$$99999x+99998y=b_2.$$

The following are computed values of x and y for different choices of b_1 and b_2.

$$b_1=200000, \quad x=200000,$$
$$b_2=200000, \quad y=-200000,$$

$$b_1=199990, \quad x=199990,$$
$$b_2=199990, \quad y=-199990,$$

$$b_1=200010, \quad x=200010,$$
$$b_2=200010, \quad y=-200010.$$

This seemingly regular behavior of the solution misleads us to the conclusion that the problem is stable and that the computer solutions are reliable. To see just how badly wrong this conclusion is, consider the totality of all solutions of the linear system of equations corresponding to all possible choices of b_1 and b_2 in the following range which contains the values of b_1 and b_2 already prescribed.

$$199990 \leqq b_1 \leqq 200010,$$
$$199990 \leqq b_2 \leqq 200010.$$

The totality of solutions which correspondingly arise are

$$-1\,800\,000 \leqq x \leqq 2\,200\,000,$$
$$-2\,200\,000 \leqq y \leqq 1\,800\,000.$$

Moreover these bounds are sharp. This set contains the solution $x=y=1$, which is obtained for $b_1=199999$, $b_2=199997$.

In numerical mathematics, the so-called condition number is often used to calibrate the sensitivity of a problem to input data. A large condition number characterizes a highly sensitive problem, while a small condition number characterizes a stable problem. Expressions for condition numbers are developed for many classes of numerical problems. For most problems computation of the condition number is as difficult as computation of the solution of the problem itself. For such problems employing the condition number is not a practical method for dealing with the accuracy of a computed solution. For some linear matrix problems, so-called cheap condition estimators are known to be useful for error estimation if some care is taken [11], [27], [28].

Recall now our earlier remark that the appearance of the first electronic computers in the early fifties, i.e., the step into the computer age, meant a thousandfold gain in speed (10^3). The actual computer revolution, however, happened afterwards. The fastest computers today are able to execute of the order of 300 million (3×10^8) floating-point operations in a second.

This is a gain in speed by a factor of 10^7 over the electronic computers of the early fifties. Compared with a person working with a mechanical desk calculator or pocket calculator of today, this is a gain in speed of the order of 10^{10}. See Fig. 5. To help grasp the significance of this factor, consider the following illustration. The human population is about 5×10^9. So, if we equip every man, woman and child with a mechanical desk calculator or an electronic pocket calculator, they could, while they are all working, perform as many operations as only one of today's faster computers.

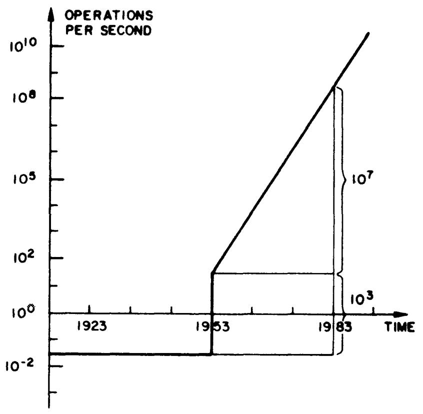


FIG. 5. *The increase in computing speed.*

We now return to our consideration of the error analysis of the computational process. The theoretical methods of backward or forward error analysis discussed earlier translate into 300 million error estimates having to be carried out for each second of a computational process. Additionally, the propagation of these errors through a complicated algorithm has to be studied. These techniques are no longer in balance with the extremely enlarged speeds of today's computers. On the other hand, the more pragmatic methods 1, 2 and 3 were all crude and finally unreliable.

In other words, when the capability of computers was relatively modest, the calculation could somehow be controlled by the user. The users were small in number, they were relatively sophisticated and they could hand-tune their computations. Today, problems which are dealt with have become enormously large and ramified, and the body of computer users comes with members of every degree of experience and sophistication. It is simply no longer possible to expect computers to be controlled by hands-on methods. There remains no alternative but to *furnish the computer with the capability of control and validation of the computational process.*

The advanced theory of computer arithmetic [18], [19] offers an approach to this question. As motivation for advanced computer arithmetic, consider a system of linear equations with coefficients that are representable in the computer without rounding errors. Then all information needed for the correct solution of the problem is present in the computer. If the problem is ill-conditioned, it may happen (as we saw earlier by means of simple examples) that the computed result has little to do with the correct solution of the problem. This means that information which was originally present in the computer has been lost by computation. The roundings are responsible for it. The act of rounding which accompanies each floating-point operation typically discards some digits. We may say that each rounding means a loss of information.

Then *the guiding principle of an advanced computer arithmetic and error analysis is to reduce the number of roundings in any particular computational process.* A central question remains: Which roundings can be omitted and which cannot?

The basic feature of advanced computer arithmetic is to augment the operator set $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$ for floating point numbers by another operation $\boxdot$ which turns out to be fundamental. $\boxdot$ is the floating-point implementation of the inner or dot product (or scalar product) of two vectors. Consistent with the implementation requirement of maximum accuracy for the four basic operations, the new scalar product must be implemented with maximum accuracy as well, i.e., with only one rounding. So, if $a=(a_1,a_2,\cdots,a_n)$ and $b=(b_1,b_2,\cdots,b_n)$ are two n-dimensional floating-point vectors, the scalar product must be defined by

$$a \boxdot b := \square\left(\sum_{i=1}^{n} a_i \times b_i\right) = \square(a_1\times b_1 + a_2\times b_2 + \cdots + a_n\times b_n)$$

for all vectors and all relevant dimensions.

Augmenting the floating-point operator set in this manner goes a long way toward controlling the loss of information inherent to floating-point calculations. The theory of computer arithmetic shows that with the augmented set of five floating-point operations, all arithmetic operations of the most customary linear spaces of computation can be performed with maximum accuracy. These spaces consist of the floating-point representations of the real and complex numbers, of the vectors and matrices over these representations and of the interval spaces over all of these.

After the four basic operations $\boxplus$, $\boxminus$, $\boxtimes$ and $\boxslash$, the linear space operations, such as the product of two matrices or the product of a matrix by a vector, are the most fundamental operations in numerical analysis. The augmented set of five basic floating-point operations, $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$ and $\boxdot$ is sufficient for the execution with maximum accuracy of these linear space operations.

Since these linear space operations are expressible in terms of scalar products, the five basic operations are in a sense necessary as well. We may expect that this enlarged set of maximally accurate computer operations, consisting of linear space operations and their interval counterparts will lead to better results in numerical computations. The enlarged set of operations support yet another fundamental feature essential for high accuracy in computation. The availability of exact scalar products, as well as matrix and matrix-vector operations with maximum accuracy, make it possible to apply a special mathematical technique in many cases, the so-called defect correction process. This process is often of scalar product type. Information that has already been lost by rounding effects during an initial computation can often be recovered by defect correction. Such corrections can be made to maximize floating-point accuracy, and in principle, they can provide arbitrary accuracy. The corresponding interval operations permit guarantees for these highly accurate results to be obtained also. Combining these two techniques within a fixed-point iteration framework, allows us to append a so-called verification or validation process to the computation. This process supplies a set of bounds for the solution to the problem being computed. Moreover, the computer delivers a proof of the existence and uniqueness of the solution of the problem within the computed bounds by verifying the hypotheses of an appropriate fixed-point theorem. We refer to the bounds and the existence proof as computer generated guarantees for the problem, simply as guarantees.

For particularly difficult problems, the validation process may not terminate within a specified time limit or iteration number limit. In this case, a warning is given to the user. Modification of the solution method is then in order.

These general techniques can be applied to fundamental problems of linear algebra, such as solving linear systems of equations, matrix inversion, polynomial or arithmetic expression evaluation, eigenvalue-eigenvector computation and linear optimization. These problems are usually solved with maximum accuracy and guarantees. This capability for these problems can be interpreted as providing additional high order arithmetic operations. Experience has shown that these methods work well even for highly ill-conditioned problems. For profoundly ill-conditioned problems, the system may fail to produce a result. In this case, notification is supplied to the user.

The reader should contrast this methodology with customary numerical practice, which only makes use of elementary computer arithmetic, that is, the four basic operations $\boxplus$, $\boxminus$, $\boxtimes$ and $\boxslash$. Results, which are supplied, are often good, but they can also be bad, even arbitrarily so. Usually no information about bounds, existence or uniqueness is provided by the conventional computation. This concludes our brief preview and motivation of advanced computer arithmetic. A more detailed discussion is given in the chapters which follow.

4. Advanced computer arithmetic. In this chapter, we deal with computer arithmetic in higher mathematical spaces (product spaces) such as spaces of complex numbers, of real and complex vectors, of real and complex matrices, of real and complex intervals, as well as the spaces of real and complex interval vectors and interval matrices. Arithmetic operations in computer representable subsets of these spaces are defined by a general mathematical mapping principle which is called a semimorphism. These arithmetic operations are distinctly different from the customary ones which are based on elementary computer arithmetic.

To make the differences clear, we begin with a brief review of the customary operations. Computers built for scientific computation are customarily equipped with the four floating-point operations $\boxplus$, $\boxminus$, $\boxtimes$ and $\boxslash$. Sometimes the eight additional corresponding operations which employ the monotone downwardly directed roundings $(\triangledown\!\!\!\!\!{+}, \triangledown\!\!\!\!\!{-}, \triangledown\!\!\!\!\!{\times}, \triangledown\!\!\!\!\!{/})$ and the monotone upwardly directed roundings $(\triangle\!\!\!\!\!{+}, \triangle\!\!\!\!\!{-}, \triangle\!\!\!\!\!{\times}, \triangle\!\!\!\!\!{/})$ are also provided. In the higher mathematical spaces, which we listed in the previous paragraph, arithmetic operations are performed by evaluating well-known mathematical formulas for them in terms of the given elementary floating-point operations (four or twelve in number, as the case may be).

For instance, if $\alpha = \alpha_1 + i\alpha_2$ and $\beta = \beta_1 + i\beta_2$ are two complex floating-point numbers or $a = (a_1, a_2, \cdots, a_n)$ and $b = (b_1, b_2, \cdots, b_n)$ are two vectors of floating-point numbers, the following product formulas are well known.

$$\alpha\times\beta=\alpha_1\times\beta_1-\alpha_2\times\beta_2+i(\alpha_1\times\beta_2+\alpha_2\times\beta_1),$$
$$a\cdot b=a_1\times b_1+a_2\times b_2+\cdots+a_n\times b_n.$$

Their computer approximations are now defined by rewriting these formulas in terms of the given floating-point operations, i.e.,

$$\alpha\boxtimes\beta=\alpha_1\boxtimes\beta_1\boxminus\alpha_2\boxtimes\beta_2+i(\alpha_1\boxtimes\beta_2\boxplus\alpha_2\boxtimes\beta_1),$$
$$a\boxdot\beta=a_1\boxtimes b_1\boxplus a_2\boxtimes b_2\boxplus\cdots\boxplus a_n\boxtimes b_n.$$

In §2 we showed, by means of simple examples, that the computational error associated with these expressions may become quite large and that this error depends critically on the given data.

Let us now make a tabulation of these higher spaces of computation. In addition to the integers, numerical algorithms are usually defined in the space (set) $\mathbb{R}$ of real numbers and vectors $V\mathbb{R}$ and matrices $M\mathbb{R}$ over the real numbers. The corresponding complex spaces $\mathbb{C}$, $V\mathbb{C}$ and $M\mathbb{C}$ also occur. All these spaces are ordered with respect to the order relation $\leqq$. In all product sets (for us all sets other than $\mathbb{R}$), the order relation is defined componentwise. The order relation is a partial order. Using the order relation $\leqq$, the notion of intervals can be defined in all these spaces. If $a\leqq b$, an interval $[a,b]$ is the set of all elements between them. That is $[a,b]:=\{x\mid a\leqq x\leqq b\}$. If we denote the set of intervals over an ordered set $\{M,\leqq\}$ by IM, we obtain the spaces $I\mathbb{R}$, $IV\mathbb{R}$, $IM\mathbb{R}$ and $I\mathbb{C}$, $IV\mathbb{C}$, $IM\mathbb{C}$. See the second column in Fig. 6.

1		2		3
		$\mathbb{R}$	$\supset$	R
		$V\mathbb{R}$	$\supset$	VR
		$M\mathbb{R}$	$\supset$	MR
$\mathbf{P}\mathbb{R}$	$\supset$	$I\mathbb{R}$	$\supset$	IR
$\mathbf{P}V\mathbb{R}$	$\supset$	$IV\mathbb{R}$	$\supset$	IVR
$\mathbf{P}M\mathbb{R}$	$\supset$	$IM\mathbb{R}$	$\supset$	IMR
		$\mathbb{C}$	$\supset$	CR
		$V\mathbb{C}$	$\supset$	VCR
		$M\mathbb{C}$	$\supset$	MCR
$\mathbf{P}\mathbb{C}$	$\supset$	$I\mathbb{C}$	$\supset$	ICR
$\mathbf{P}V\mathbb{C}$	$\supset$	$IV\mathbb{C}$	$\supset$	$IVCR$
$\mathbf{P}M\mathbb{C}$	$\supset$	$IM\mathbb{C}$	$\supset$	$IMCR$

FIG. 6. *Table of spaces occurring in numerical computations.*

Most algorithms in numerical analysis are defined in one or several of these spaces. However, these algorithms cannot usually be executed in these spaces. For execution, we use computers. A computer contains only a subsystem R of the real numbers. R is the set of computer reals or floating-point numbers. (Sometimes several such systems of differing precision are available.) Vectors (n-tuples), matrices ($n\times n$-tuples), complexifications (pairs), vectors and matrices of such pairs, as well as the corresponding sets of intervals, can be defined in terms of R. Doing so, we obtain the spaces VR, MR, IR, IVR, IMR, CR, VCR, MCR, ICR, $IVCR$ and $IMCR$, which are listed in the third column of Fig. 6. We indicate set-subset relations in Fig. 6 by means of inclusion symbol $\supset$.

Having described the sets listed in the third column of Fig. 6, we turn to the arithmetic operations to be defined for these sets. Our definition of these operations is essentially different from the conventional one. These operations are supposed to approximate the operations in the corresponding sets listed in the second column. The operations are well known in any of the spaces $\mathbb{R}$, $V\mathbb{R}$, $M\mathbb{R}$, $\mathbb{C}$, $V\mathbb{C}$ and $M\mathbb{C}$ of the second column. The powerset $\mathbf{P}M$ of any set M is defined as being the set of all subsets of M. The powersets of the sets just enumerated are listed in the first column of Fig. 6. Now if $*$ is any operation defined in M, then a corresponding operation $*$ can be defined in the powerset $\mathbf{P}M$ as follows.

$$A * B := \{a * b \mid a \in A \wedge b \in B\} \quad \text{for all } A,\ B \in \mathbf{P}M.$$

This definition extends every operation of M into the corresponding powerset $\mathbf{P}M$. Summarizing, we can now say that the operations in the sets listed in the leftmost element of every row in Fig. 6 are always known. Of course, all of these operations are ideal mathematical operations. We now use these ideal operations to define operations in the subsets on the right-hand side of Fig. 6, row by row, using a general mapping principle.

Let M denote any set of Fig. 6 in which the operations are known and N the subset on its right in the same row. For each $*$ in M, we define an operation $\boxed{*}$ in N as follows:

(RG) $$a \boxed{*} b := \square(a * b) \quad \text{for all } a,\ b \in N \text{ and for all } *.$$

Here $\square$: $M \to N$ denotes a mapping with the following properties.

(R1) $\square a = a$ for all $a \in N$ (rounding).

(R2) $a \leqq b \Rightarrow \square a \leqq \square b$ for all $a,\ b \in M$ (monotonicity).

(R4) $\square(-a) = -\square a$ for all $a \in M$ (antisymmetry).

In the case of the interval sets of Fig. 6, the order relation $\leq$ means set inclusion $\subseteq$. In this case, we also require that the rounding $\square$ has the property

(R3) $a \leqq \square a$ for all $a \in M$ (upwardly directed).

Property (R3) is referred to as the property of isotony of the rounding $\square$. In mathematical settings, a set with operations is sometimes considered where, in fact, the operations are seemingly not executable. Mathematicians then usually look for another set with executable operations and try to arrange an isomorphism between the two sets and corresponding operations. Isomorphism is the strongest relevant mathematical mapping principle. It has the property that the inverse image of the result of a computation in the image set is the result that would have been obtained if the computation could have been executed in the original set.

Since the operations in the leftmost element of each row of Fig. 6 are not computer implementable, we have a situation of the type just described. However, in Fig. 6 set-subset pairs occur which are of different cardinality, and isomorphisms cannot be established between such sets.

A somewhat weaker mathematical mapping principle is that of a homomorphism. It can be shown by simple examples [19] or by a theorem that even homomorphisms cannot be established between the relevant sets in Fig. 6.

Now we can try to weaken the mapping properties of a homomorphism still further. Doing so, we reach a mapping correspondence with our properties (RG), (R1), (R2), (R3), (R4). These properties can be derived as necessary conditions for a homomorphism between ordered algebraic structures [19]. Therefore, we call the mapping, which they characterize, a semimorphism. The mapping principle of semimorphism between relevant sets in Fig. 6 seems to be as far as we can go toward homomorphism. We also define the outer operations that occur in Fig. 6 (scalar times vector, matrix times vector, etc.) by corresponding semimorphisms.

It is important to understand that the arithmetic operations for the product sets defined by semimorphism are different in general from those which arise if only elementary floating-point arithmetic is furnished. Semimorphism defines operations in a subset N of a set M directly by making use of the operations in M. It makes a direct link between an operation in M and its approximation in the subset N. For instance, the operations in MCR (see Fig. 6) are directly defined by the operations in $M\mathbb{C}$, and not in a round about way via $\mathbb{C}, \mathbb{R}, R$, CR and MCR as it would have to be done by using the elementary arithmetic only.

It is easy to see that repetition of semimorphism is again a semimorphism. The operations in the leftmost element of every row in Fig. 6 are all well known. This allows us to define operations in all sets of Fig. 6 by semimorphism. As already noted, the outer operations in Fig. 6 are defined by semimorphism also.

The new operations now defined in all sets of Fig. 6 are of an accuracy which we call maximal for all admissible roundings. This means that between the correct result of an operation and its approximation in the subset no other element of the subset may be found [19]. Maximal accuracy guarantees the result to be within one unit in the last place. This fundamental result follows readily from (RG), (R1) and (R2). For instance, in the case of multiplication of two real or complex floating-point matrices $a=(a_{ij})$ and $b=(b_{ij})$, (RG) means

$$a \boxdot b := \square(a \times b) = \square\left(\sum_{k=1}^{n} a_{ik} \times b_{kj} \right).$$

Here the rounding is defined componentwise. That is, there occurs only one single rounding error in each component of the product matrix, even for very large n. Compare this with the result that is obtained when only elementary floating-point arithmetic is used. Earlier we saw that in such a case, 8 million roundings (each equivalent to a loss of information) are performed in a multiplication of two 100×100 complex floating-point matrices. Compare this first to one rounding in each of only 10,000 components of the result. Compare this secondly with only a single rounding altogether in terms of the space MCR where the multiplication is actually defined. This number one is not a fiction since no matrix in MCR lies between the exact result and the result computed by the new semimorphic operation. These new operations are not only much more accurate, they are of a simpler form as well. Thus, they allow a simpler error analysis of numerical algorithms, and they lead to more accurate error estimates and bounds. All of this leads to a control of errors in computation by the computer itself as we shall see.

4.1. Comments on the derivation of semimorphisms. We have noted that certain essential properties of a semimorphism can be obtained as necessary conditions for a homomorphism between ordered algebraic structures. There are still other possibilities of deriving the properties of a semimorphism. They can be derived directly by considering special models of sets in Fig. 6. For instance, consider the mapping of the powerset of the complex numbers **PC** into the intervals of the complex numbers $I\mathbb{C}$. An interval $[a, b]$ of two complex numbers a and b with $a \leqq b$ is a rectangle in the complex plane.

If we multiply two complex intervals A and B in the sense of the powerset operation (see Fig. 7), we do not generally obtain an interval result, but a more arbitrary element $A \times B$ of the powerset $\mathbf{P}\mathbb{C}$.

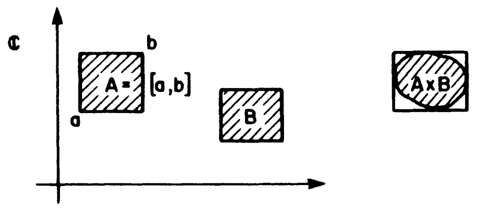


FIG. 7. *Multiplication of complex intervals.*

We require that the result of an interval operation be an interval. The best we can do is to map the powerset product $A \times B$ onto the least interval that contains it. In Fig. 7 this is shown as a rectangle fitting snugly around the set $A \times B$. It is not difficult to see that this mapping $\square$: $\mathbf{P}\mathbb{C} \to I\mathbb{C}$ is a rounding having all of the properties, (R1, 2, 3, 4) and (RG) of a semimorphism. In the present case, the order relation is set inclusion $\subseteq$. If the set $A \times B$ is already an interval, the rounding has no effect, i.e., (R1) holds. If we enlarge the set $A \times B$ somewhat, this enlarges the least interval that includes it also, i.e., (R2) holds. (R3) is obvious and (R4) holds by reasons of symmetry. The result fulfills (RG) viz, $A \boxtimes B = \square(A \times B)$, by construction.

As a second example, consider the basic pair $\mathbb{R}$ and R. If we add two floating-point numbers a and b (row 1), then the correct sum $c = a + b$ is not in general a floating-point number (Fig. 8). To obtain a floating-point number, we round the result into the floating-point screen. Referring to Fig. 8 one can see that the process of rounding fulfills (R1), (R2) and (R4). In this case, the order relation is $\leqq$. The floating-point operation is defined by (RG): $a \boxplus b = \square(a + b)$.

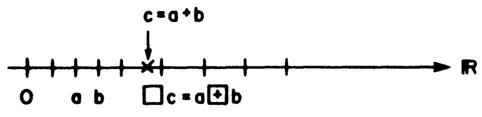


FIG. 8. *Floating-point addition.*

Simple intuitive pictures of the operations are not available for all pairs of relevant spaces in Fig. 6. The computer structures in the third column of Fig. 6 are examples of certain basic mathematical structures, the so-called ordered or weakly ordered ringoids and vectoids. These basic structures are invariant under semimorphism. These concepts and properties cannot be discussed here. It turns out that they are very useful for deriving computer implementable algorithms for many operations [19].

In the case of the mapping of the real numbers into the floating-point numbers, it is very useful to provide two additional roundings and the arithmetic operations associated with them. These two roundings are the monotone directed roundings ∇ and $\triangle$. They are defined by (R1), (R2) and

(R3) $$\nabla a \leqq a \text{ and } a \leqq \triangle a \text{ for all } a \in M.$$

The corresponding operations are defined by

(RG) $$a \overset{*}{\nabla} b := \nabla(a*b) \quad \text{and} \quad a \overset{*}{\triangle} b := \triangle(a*b)$$
$$\text{for all } a,\ b \in N \text{ and all } * \in \{+,-,\times,/\}.$$

The monotone downwardly directed rounding ∇ maps the entire interval between two neighboring floating-point numbers onto the lower bound of this interval. The monotone upwardly directed rounding $\triangle$ maps such an interval onto its upper bound.

4.2. Implementation of advanced arithmetic on computers. Having used semimorphism (property (RG)) to define all of the many floating-point arithmetic operations associated with Fig. 6, we next deal with the implementation of these operators on computers. We seek implementations by means of fast algorithms. The resulting methods are comparable in speed to implementations corresponding to operations based on elementary computer arithmetic only.

At first sight it seems doubtful that formula (RG) can be implemented on computers at all. To determine the approximation $a \boxed{*} b$, the result $a*b$ seems to be called for. In general, $a*b$ will not be representable and a fortiori not executable on the computer.

Thus $a*b$ is not available for implementation of $a \boxed{*} b$. We use isomorphic relationships to deal with this problem. It can be shown that for the operations defined by semimorphism (RG), there exist isomorphic computer executable representations in all cases of Fig. 6. A detailed analysis of this question is given in [19]. For the experienced reader, we offer some comments about this question.

Formula (RG) defines computer operations. While showing that there exist isomorphic representations of the computer representable subsets, only the structure of these subsets and their operations in N defined by semimorphism may be used. Therefore, a careful analysis of this structure has to made in advance. This structure is different from the one in M with which mathematicians usually work.

The theory of computer arithmetic shows that all operations that occur in the third column of Fig. 6 can be realized by a modular technique. This calls for a module where the following fifteen operations are made available on the computer. These operations

$$\begin{array}{ccccc} \boxed{+} & \boxed{-} & \boxed{\times} & \boxed{/} & \boxed{\cdot} \\ \overset{+}{\nabla} & \overset{-}{\nabla} & \overset{\times}{\nabla} & \overset{/}{\nabla} & \overset{\cdot}{\nabla} \\ \overset{+}{\triangle} & \overset{-}{\triangle} & \overset{\times}{\triangle} & \overset{/}{\triangle} & \overset{\cdot}{\triangle} \end{array}$$

are sufficient for the computer implementation of all operations that occur in the third column of Fig. 6. We shall comment on the remaining part of the implementation question in §6. Here $\boxed{*}$, $* \in \{+,-,\times,/\}$ denotes the semimorphic operations defined by (RG), using some particular monotone and antisymmetric rounding (R1, R2, R4), such as rounding to the nearest number of the screen. Likewise $\overset{*}{\nabla}$ respectively $\overset{*}{\triangle}$, $* \in \{+,-,\times,/\}$ denote the operations defined by (RG) and the monotone downwardly respectively upwardly directed rounding. $\boxed{\cdot}$, $\overset{\cdot}{\nabla}$, and $\overset{\cdot}{\triangle}$ denote three scalar products with maximum accuracy. That is, if $a=(a_i)$ and $b=(b_i)$ are vectors of floating-point numbers, then

$$a \odot b := \bigcirc(a_1 \times b_1 + a_2 \times b_2 + \cdots + a_n \times b_n) \quad \text{for all } \bigcirc \in \{\square, \nabla, \triangle\}.$$

The multiplication and addition signs on the right-hand side of the expression denote exact multiplication and summation in the sense of real numbers. Comments on the implementation of the operations $\boxplus$, $\boxminus$, $\boxtimes$, and $\boxed{/}$ were already given in §2. It is useful to provide the three scalar products, $\boxdot$, $\nabla\kern-0.6em\cdot\kern0.3em$, $\triangle\kern-0.6em\cdot\kern0.3em$, in two different modes: the first one adding to an initial value zero and the second one adding to the result of a preliminary and unrounded scalar product. The second mode makes it possible to evaluate sums of scalar products $\square(u\cdot v+x\cdot y)$ or sums of matrix products $\square(A\cdot B+C\cdot D)$with only one rounding at the end of the computation. These modes of the scalar product are key requirements for the defect correction process which is used in the self-validating computational procedures which we shall treat in §5.

Of these 15 fundamental operations above, traditional numerical methods use only the four operations $\boxplus$, $\boxminus$, $\boxtimes$, and $\boxed{/}$. Interval arithmetic employs the eight operations $\nabla\kern-0.65em{\scriptstyle +}$, $\nabla\kern-0.65em{\scriptstyle -}$, $\nabla\kern-0.65em{\scriptstyle \times}$, $\nabla\kern-0.65em{\scriptstyle /}$ and $\triangle\kern-0.65em{\scriptstyle +}$, $\triangle\kern-0.65em{\scriptstyle -}$, $\triangle\kern-0.65em{\scriptstyle \times}$, $\triangle\kern-0.65em{\scriptstyle /}$. These eight operations are computer equivalents of the operations for real intervals, i.e., of interval arithmetic. The recently proposed arithmetic of the so-called IEEE standard offers 12 of these 15 fundamental operations: $\boxed{*}$, $\nabla\kern-0.65em{\scriptstyle *}$, $\triangle\kern-0.65em{\scriptstyle *}$, $*\in\{+,-,\times,/\}$ [10], [15]. Roughly speaking, interval arithmetic brings guarantees into computation while the three scalar products deliver high accuracy. These two features should not be confused. We return to these matters in §5.

4.3. Implementation of scalar products. Because of the importance of optimal scalar products, we comment on their implementation on computers. Such implementation should be made by means of fast hardware routines. A black box technique is used where the components a_i and b_i, $i=1(1)n$, are the input and the scalar products with maximum accuracy $\boxdot$, $\nabla\kern-0.6em\cdot\kern0.3em$, $\triangle\kern-0.6em\cdot\kern0.3em$ the output. See Fig. 9. The black box requires a local store. The size of the local store depends on the data formats in use (number system base, mantissa length and exponent range). In particular, the size is essentially independent of the dimension n of the two vectors $a=(a_i)$ and $b=(b_i)$ to be multiplied.

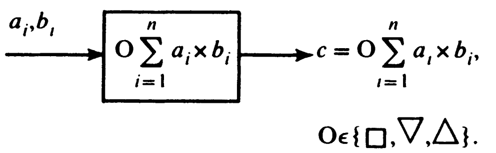

a_i, b_i
$\bigcirc\sum_{i=1}^{n} a_i\times b_i$
$c=\bigcirc\sum_{i=1}^{n} a_i\times b_i,$
$\bigcirc\in\{\square,\nabla,\triangle\}.$

FIG. 9. *The black box for scalar products.*

Access to the local store should be much faster than access to main storage. The full product $a_i\times b_i$ is required. This mandates a mantissa of $2l$ digits and an exponent range $2e1\leqq e\leqq 2e2$. This reduces the problem to an implementation of the sum

$$\bigcirc\sum_{i=1}^{n} c_i,\qquad \bigcirc\in\{\square,\nabla,\triangle\}$$

on the computer. Here the c_i, $i=1(1)n$, denote floating-point numbers of $2l$ digit mantissas, i.e., $c_i\in R(b,2l,2e1,2e2)$.

If one of the summands c_i has exponent 0, its mantissa can be expressed in a register of length $2l$. If another summand has exponent 1, it can be expressed with exponent 0, if the register provides further digits on the left and the mantissa is shifted one place to the left. An exponent -1 in one of the summands requires a corresponding shift to the right. The largest exponents in magnitude that may occur in the summands c_i are $2e2$ and $2|e1|$. This shows that all summands can be expressed (in a type of fixed-point representation) in a register of length $2e2+2l+2|e1|$ without loss of information. If the register is built as an accumulator, all summands could even be added without loss of information. In order to accommodate possible overflows, it is convenient to provide a few, say t more digits of base b on the left. In such an accumulator, every such sum or scalar product can be added without loss of information. As many as b^t overflows may occur and be accounted for without loss of information. In the worst case, presuming every sum causes an overflow, we can accommodate sums with $n \leqq b^t$ summands.

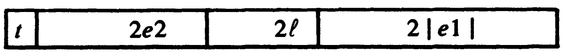


FIG. 10. *Register for scalar product accumulation.*

Actually the superlong accumulator may be replaced by a local store of size $d=t+2e2+2l+2|e1|$ and an adder of size approximately $3l$. The summands are all of length $2l$. The local store is organized in words of length l. Since the summands are of length $2l$, they fit into a part of length $3l$ of this local store. This part of the store is determined by the exponent of the summand. We load this part of the store into an accumulator of length $3l$. The summand mantissa is placed in a shift register and is shifted to the correct position as determined by the exponent. Then the shift register contents are added to the contents of the accumulator. Instead of the shift register in Fig. 11, a cross point switch may be used to achieve a faster parallel implementation.

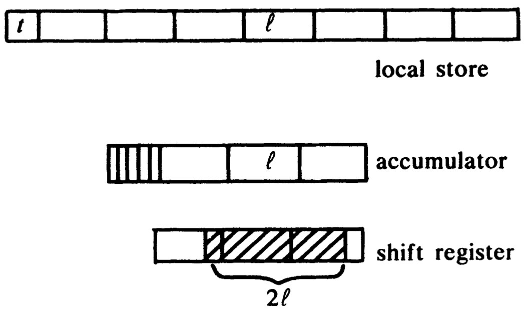


FIG. 11. *Addition process for scalar products.*

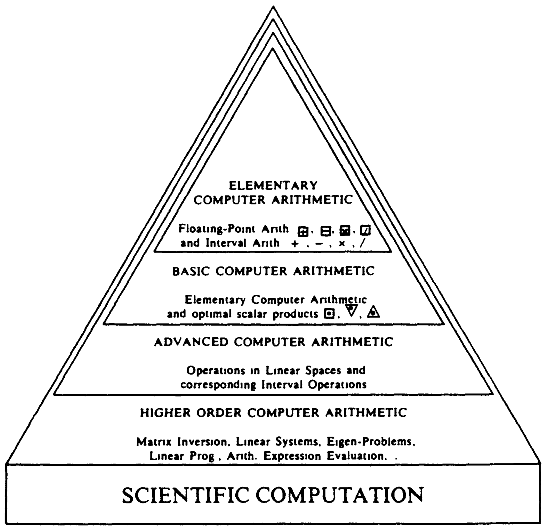
An addition into the accumulator may produce a carry. To accommodate carries, we enlarge the accumulator on its left end by a few more digit positions. These positions are filled with the corresponding digits of the local store. If not all of these digits equal $b-1$, they will accommodate a possible carry of the addition. Of course, it is possible that all these additional digits are $b-1$. In this case, a loop has to be provided that takes care of the carry and adds it to the next digits of the local store. This loop may need to be traversed several times.

Other addition techniques or carry handling processes are possible. While the addition process described was in terms of hardware registers, it can, of course, also be simulated in software. For a more detailed discussion of these principles, see [8] [9], [20]. Special purpose long accumulators have appeared earlier in computer design [22].

Independent of questions of accuracy, conventional computation of the scalar product using elementary floating-point arithmetic only is a far slower process than the one just described. Some reasons for this are:

1. The optimal scalar product algorithm can locate a subsequent operand simply by increasing the current address, thus avoiding complicated index calculations or optimization techniques.
2. Some data transports to and from the stack and some range checks are avoided.
3. The average shift of the summands into proper position in the shift register of length $3l$ is shorter than in the case of a standard addition technique.
4. The main step in a scalar product computation: $s := s + a \times b$ contains one addition and one rounding. This rounding, as well as a normalization step, are avoided by the original algorithm.

Figure 12 illustrates the steps of development of arithmetic as a basis for scientific computation. The first three levels, Elementary Computer Arithmetic, Basic Computer Arithmetic and Advanced Computer Arithmetic have now been discussed. In the next chapter we deal with the extension of ideas and capability to so-called Higher Order Computer Arithmetic, namely to the fundamental algorithms of numerical analysis, that is to matrix inversion, linear system solving and so forth.


ELEMENTARY COMPUTER ARITHMETIC
Floating-Point Arith ⊞, ⊟, ⊠, ⊘ and Interval Arith +, −, ×, /
BASIC COMPUTER ARITHMETIC
Elementary Computer Arithmetic and optimal scalar products ⊡, ▽, △
ADVANCED COMPUTER ARITHMETIC
Operations in Linear Spaces and corresponding Interval Operations
HIGHER ORDER COMPUTER ARITHMETIC
Matrix Inversion, Linear Systems, Eigen-Problems, Linear Prog., Arith. Expression Evaluation. .
SCIENTIFIC COMPUTATION

DEVELOPMENT OF COMPUTER ARITHMETIC

BROADENING THE BASE FOR SCIENTIFIC COMPUTATION

FIG. 12

5. Extension to self-validating methods. In §4, we noted that computer realization of the five basic operations $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\odot$ for each of three different roundings $\bigcirc \in \{\square, \nabla, \triangle\}$ suffices for an implementation of the arithmetic operations of maximal accuracy in the commonly used linear spaces of scientific computation and their interval correspondents. In this chapter we indicate how these operations are applied to fundamental problems in linear algebra such as linear systems of equations, eigenvalue-eigenvector computation and optimization problems. Such problems can usually be solved with high, even maximum accuracy, even in severely ill-conditioned cases. Furnishing the solution of these problems with maximal accuracy allows the process of their solution to be interpreted as additional high order arithmetic operations.

Roughly speaking, it is the scalar products with maximal accuracy together with other techniques which deliver the high accuracy of these higher order operations, while it is interval arithmetic which delivers guarantees. Taken together, these features provide the basis for an automatic validation process for the computation being performed. Interval arithmetic has been used in numerical mathematics for about 25 years, and highly sophisticated methods have been developed [1], [2], [22], [23], [S10]. An extensive list of references to interval mathematics can be found in [2]. Interval arithmetic has often been criticized since its naive use may deliver bounds which are unreasonably large, and thus, do not contain much information about the solution of the problem. This has been interpreted as a failure of interval arithmetic for delivering high accuracy. This criticism is fundamentally misdirected. Interval arithmetic is the only computational tool so far available that incorporates guarantees as part of the basic computational process. It is very useful for this purpose. High accuracy is not intrinsic to interval arithmetic. High accuracy is obtained by use of the process of residual correction. There are well-known limitations to residual correction. It is our use of the optimal scalar product which makes residual correction effective. It is the combination of these two features, namely, interval arithmetic and residual correction with the optimal scalar product which delivers guarantees with high or even maximum accuracy.

We give our discussion of an extension of advanced computer arithmetic to fundamental problems of numerical analysis with an informal description of how these methods work. As a first step, an initial approximation to the solution of the problem is computed by some favored method such as Gaussian elimination in the case of a linear system. The quality of this initial guess is occasionally critical for the success of the next step. A second and basic step is to cast the problem to be solved into an iterative solution process. The iteration is pursued in the so-called residual correction mode. This is a a well-known computational technique in which, at each iterative step, the residual or defect in the current approximation is computed and used to modify the approximation. Here the precise computer arithmetic and in particular, the precise scalar products come into play. As the quality of the approximation iteratively improves and the values of the defects or residuals diminishcorrespondingly, further correction by this process becomes critical. To refine something already of high quality requires the use of more highly refined attributes. The accurate approximation can only be made more accurate with an ever more accurate calculation of the residual. Less figuratively, we note that as the correction process continues, the successive residuals tend toward zero. Thus, computation of the residual is characterized by an increasing degree of cancellation. Then to achieve any particular number of leading digits in the determination of a residual requires increasingly more accuracy in its computation. The relative accuracy needed does, in fact, depend on the problem, and for some problems, it can be quite large. In technical terms, ill-conditioning of a problem grades the degree of accuracy needed in residual computation for correction of the current iterate. This is the point where the optimal scalar product plays a critical role. By using optimal scalar products in the adding mode, residuals may be calculated to all relevant figures for correction. This is the basic process, by means of which high accuracy is obtained. By contrast, if only elementary floating-point arithmetic is used, any given precision (single, double, extended) in which this calculation might be performed, could prove to be inadequate for correction of the iterate beyond a certain point. This point is quite vulnerable to the ill-condition of the problem, the latter in general unknown.

How do we obtain verified results? This is where interval arithmetic, along with the optimal dot products $\triangledown$ and $\triangle$ with the directed roundings (upwards and downwards) and their interval equivalents play their critical role. At some point in the iteration, the mode of computation is switched, so that the data types being employed become intervals. This point is chosen adaptively using a criterion which detects saturation in the pointwise process. Thereafter, the residual correction process is continued using interval arithmetic including optimal scalar products. In the traditional use of naive interval arithmetic, this could be a counter-productive step, since as is well known, naive interval arithmetic often swells the size of the intervals which it handles.

Here a new and important feature is brought into play. The numerical process of residual correction is a contracting process in general. This mathematical term means that the distance between any two data is reduced by each step of the process. Indeed, it is this property which makes the iterative methods of numerical analysis work. We now play this contraction against the tendency of the swelling of the interval arithmetic. Here once again, use is made of the optimal (interval) scalar product. In practice the contraction is able to dominate. The resulting process is a residual correction using interval arithmetic where the intervals contract.

Finally, a last tactic is brought into play. When one of the intervals occurring in the iteration process is enclosed within its predecessor interval, the Brouwer fixed point theorem is used for validation. This theorem asserts that when a mapping, such as our iteration process with intervals, results in an interval which lies within the predecessor interval, then that mapping has a fixed point within the interval.

- The fixed point statement is equivalent to the existence of the solution.
- The contraction of the mapping provides uniqueness of the solution.
- The final interval presents a set of bounds for the solution. These bounds, if inadequately sharp, may be improved by that very same iterative correction process employing the optimal scalar products already used. Improvement to arbitrary accuracy is possible in principle.

The mapping property of contraction may be difficult to verify. In practice, therefore, other criteria for establishing uniqueness are used. One such property which is moreover computer verifiable is retraction, i.e., strict inclusion of a proper linearization of the mapping.

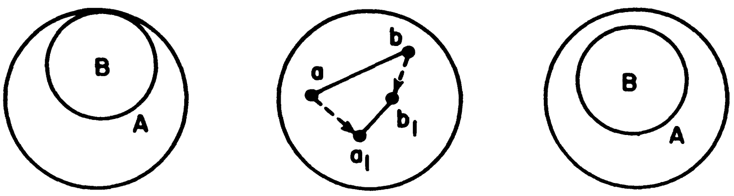
The process that establishes the three properties, displayed above, is called verification or validation of the result of the computation. It establishes the existence and uniqueness of the solution of the problem within the computed bounds. (Such methods are sometimes also called E-methods (corresponding to the German words for existence, uniqueness and containment: *Existenz*, *Eindeutigkeit*, *Einschliessung*)). We stress that the validation is an automatic process performed by the computer. The computer is not per se proving existence and uniqueness of a solution. It is simply being used to verify the hypotheses of a theorem which furnishes this proof. Methods that provide results of high accuracy with guarantees are available for many standard problems of numerical analysis, such as: linear systems of equations, matrix inversion, eigenvalue-eigenvector computation, polynomial and arithmetic expression evaluation, optimization problems, nonlinear systems of equations and even for problems with differential equations [2], [13], [14], [16], [24], [26].

In certain extremely ill-conditioned problems, the system may fail to produce a validation. In this case, the user is notified. A modification of the solution method is in order. All this is in sharp contrast to conventional numerical packages where results may be supplied in such cases which deviate arbitrarily from the exact results without a proper warning.

5.1. Imprecise data. Advanced computer arithmetic, as we describe it here, includes the operations for intervals over all the commonly used linear spaces of computation. Thus, scientific problems which themselves furnish data of limited accuracy are conveniently accommodated. Input data, which are not precisely known may be specified as an interval. In such a case, the results are intervals which are verified to contain all potential results which can arise from data values within the specified intervals. The results, of course, cannot be made more accurate than the data allow them to be, but verified results with guarantees, i.e., validation can generally be supplied.

The fact that computers can be used to provide such qualitative statements as the existence and uniqueness of the solution of a particular problem within the computed bounds by means of arithmetic computation opens a new dimension for scientific computation. Such a computer is no longer merely a fast calculating tool, but a scientific instrument of mathematics. Moreover this tool is user friendly to the naive as well as the sophisticated user. We stress once more that these results can more or less be easily obtained if the 15 fundamental operations displayed in §4 are made available on the computer. The availability of interval operations is essential for obtaining these results.

We now illustrate the three mapping principles of (a) inclusion, (b) contraction and (c) retraction, resp. which are relevant for our treatment. See Fig. 13a, b, c, resp.



a) *Inclusion* b) *Contraction* c) *Retraction*

FIG. 13.

5.2. Continuous mappings. Let f: $\mathbb{R}^n \to \mathbb{R}^n$ be a continuous mapping. Let A be a nonempty, convex, closed and bounded subset of $\mathbb{R}^n$. Let $f(A)=B$.

(a) *Inclusion*: $B \subseteq A$ establishes existence of a fixed point (Brouwer fixed point theorem).

(b) *Contraction*: Let $f(a)=a_1$, $f(b)=b_1$. If, for all a, $b \in A$, the distance $d(a_1,b_1) \leqq kd(a,b)$ with $k<1$, then uniqueness is established. To see this, suppose x and y are fixed points. Then $d(x,y) \leqq kd(x,y)$. Then $(1-k)d(x,y) \leqq 0$ so that $d(x,y)=0$. This implies that $x=y$.

(c) *Retraction*: Fig. 13c displays the case where B is a retraction of A, that is, where B lies inside of A and away from the boundary of A. Retraction implies uniqueness of the fixed point in the case of a linear mapping. To see this, suppose there are two fixed points x and y of the linear mapping f. Then for any scalar λ, $f(x+\lambda y)=f(x)+\lambda f(y)=x+\lambda y$. Then $x+\lambda y$ is a fixed point also. Now λ can be chosen such that $x+\lambda y$ is on the boundary of A. This contradicts the retraction property.

If A and B are intervals, $A=[a_1,a_2]$ and $B=[b_1,b_2]$ of any dimension, then (a) and (c) are easily testable. Indeed:

$$\begin{aligned} &(a) \quad \Leftrightarrow \quad a_1 \leqq b_1 \wedge b_2 \leqq a_2, \\ &(c) \quad \Leftrightarrow \quad a_1 < b_1 \wedge b_2 < a_2. \end{aligned}$$

Contraction (b) is a property of the mapping which must be known to the user. In the rare case of certain special mappings, the computer itself can verify the contraction.

For the specialist, we give a more formal but still brief description of the iteration and verification process described above. We choose the simple case of solving a system of linear equations of the form

$$Ax = b. \tag{1}$$

Let the exact solution be denoted by $\hat{x}$ and an approximate solution by $\tilde{x}$. Let $e = \hat{x} - \tilde{x}$ be the error and denote the defect of the approximation by $d := b - A\tilde{x}$. With the optimal scalar product in the adding mode, d can be computed with full (or at least to maximal) accuracy. Then

$$b - A\tilde{x} = d, \qquad b - A\hat{x} = 0, \tag{2}$$

and therefore,

$$Ae = d. \tag{3}$$

If we now compute an interval inclusion E for the error e, we obtain an inclusion for the solution of (1):

$$e = \hat{x} - \tilde{x} \in E \Rightarrow \hat{x} \in \tilde{x} + E.$$

Now we consider the interval iteration scheme

$$E_{n+1} := (I - RA)E_n + Rd \tag{4}$$

with any matrix R. Here I denotes the identity matrix. A theorem of interval analysis [2] tells us that (4) converges for every initial interval vector E_0 to the unique fixed point of (4) if and only if the spectral radius $\rho(|I - RA|) < 1$. Here we use the absolute value of a matrix to denote the matrix of the absolute values of its components. E_{n+1} is a retraction of E_n if

$$E_{n+1} \subset E_n \quad \text{and} \quad \operatorname{dist}(E_{n+1}, \partial E_n) > 0 \tag{5}$$

where ∂E_n denotes the boundary of E_n, $\operatorname{dist}(X, Y)$ denotes the distance between X and Y. Retraction is easy to guarantee computationally. If (5) holds, another theorem states that R and A are not singular and E_{n+1} contains the solution of (3): $e \in E_{n+1}$. Thus, the interval $\tilde{x} + E_{n+1}$ contains the unique solution of (1).

In practice R is chosen as an approximate inverse of A. Then $\rho(|I - RA|) < 1$ usually holds in practice. Now the starting value E_0 for the iteration (4) is obtained by adding a very small interval to the approximation $\tilde{x}$ of $\hat{x}$. In most cases in practice the retraction (5) occurs after one iteration. Only rarely have more iterations been needed to produce (5), and then only two or three. (4) is very sensitive to roundings. Therefore, the optimal scalar product in the adding mode is used during the computation of (4). The whole process delivers a highly accurate set of computed bounds for the solution and simultaneously proves the existence and uniqueness of the solution within these computed bounds.

(2) and (4) are suited to reveal the differences between the use of elementary and advanced computer arithmetic. Let us first assume that elementary computer arithmetic is used: If $\tilde{x}$ in (2) is already a good approximation of $\hat{x}$, then $A\tilde{x}$ is close to b and the subtraction $b - A\tilde{x}$ causes cancellation. That is, only a few digits or possibly no digit of d can be computed correctly. For an imprecisely known d, determination of the error e from (3) is worthless.

Similarly, if in (4) R is a good approximation of A^{-1}, then RA is close to I and the subtraction $I-RA$ causes cancellation. In such a case the iteration matrix in (4) is known only approximately so that the computed result of the iteration (4) is worthless.

On the other hand, by using advanced computer arithmetic with optimal scalar products, the difference $b-A\tilde{x}$ in (2) and the iteration matrix $I-RA$ in (4) can be computed to full accuracy. This makes the defect correction process work very well.

In [17] it is shown that validation is provided for a linear system in at most 6 times the work for solving the linear system itself by Gaussian elimination. This theoretical bound has proved to be pessimistic in practice, since, as we have already noted, rarely is more than one iteration required to produce the validation. Sparse matrices may be accommodated by these methods without increasing storage requirements. With effective implementation the speed of these methods has been made comparable to any alternate method and this includes the validation which is not typically provided by alternate methods.

Linear systems of equations play a central role for the whole of numerical analysis. Many problems can be reduced to linear systems. Even nonlinear systems of equations are solved approximately through the use of linear systems.

Accurate function evaluation is indispensible for many algorithms. Then as the last step of our treatment of "high order" arithmetic operations, we show how polynomials and then arbitrary arithmetic expressions can be evaluated with high accuracy (the validation step included) [4]. We proceed by reducing these questions to solving either linear systems of equations or to solving nonlinear systems of equations of special form.

5.3. Expression evaluation. As a model situation, consider the following polynomial of degree three.

$$p(t)=a_3t^3+a_2t^2+a_1t+a_0=((a_3t+a_2)t+a_1)t+a_0,$$

where a_0, a_1, a_2, a_3 and t are given floating-point numbers. The expression on the right-hand side is called the Horner scheme. Evaluation of $p(t)$ by means of the Horner scheme proceeds in the following steps:

$$\begin{array}{lcrl} x_1=a_3 & & x_1 & =a_3 \\ x_2=x_1t+a_2 & & -tx_1+x_2 & =a_2 \\ x_3=x_2t+a_1 & \text{or} & -tx_2+x_3 & =a_1 \\ x_4=x_3t+a_0 & & tx_3+x_4 & =a_0. \end{array}$$

This is a system of linear equations $Ax=b$ with a lower triangular matrix, where

$$A=\begin{pmatrix} 1 & 0 & 0 & 0 \\ -t & 1 & 0 & 0 \\ 0 & -t & 1 & 0 \\ 0 & 0 & -t & 1 \end{pmatrix}, \quad x=\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \quad \text{and} \quad b=\begin{pmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}.$$

x_4 is the value of the polynomial. Then a highly accurate solution of the linear system delivers the value of the polynomial with high accuracy. The extension to higher order polynomials is obvious. This procedure generates highly accurate evaluation of polynomials, even of very high order.

Let us now consider general arithmetic expressions and begin with the example

$$(a+b)c-\frac{d}{e}.$$

Here a, b, c, d and e are floating-point numbers. Evaluation of this expression can be performed in the following steps.

$$\begin{aligned} x_1 &= a, \\ x_2 &= x_1 + b, \\ x_3 &= cx_2, \\ x_4 &= d, \\ ex_5 &= x_4, \\ x_6 &= x_3 - x_5. \end{aligned}$$

Once again we obtain a linear system of equations with a lower triangular matrix.

There are arithmetic expressions which lead to a nonlinear system of equations. For example, the expression

$$(a+b)(c+d)$$

leads to the nonlinear system of equations

$$\begin{aligned} x_1 &= a, \\ x_2 &= x_1 + b, \\ x_3 &= c, \\ x_4 &= x_3 + d, \\ x_5 &= x_2 x_4. \end{aligned}$$

All such systems are of a special lower triangular form. They can be solved directly, or they can be transferred into linear systems by an automatic algebraic transformation process [4]. Solution techniques which employ optimal scalar products and defect correction methods can then be used. In this way the value of an arithmetic expression with high accuracy is obtained. The extension of computation with high accuracy from dyadic operations (even in the product spaces of Fig. 6) to arbitrary arithmetic expressions is fundamental. Even though the operations are implemented optimally, in computations involving several such operations errors may accumulate and become large. With optimal scalar products and defect correction methods, we can reduce the loss of information in the evaluation of polyadic operations of arithmetic expressions to only one single rounding.

6. Connection with programming languages. We had earlier noted that the stored program was a major intellectual breakthrough in the development of computing machinery. The program is the means by which the computer user lays out the work that he expects the computer to perform. In a sense he writes the requirements of his computation in a basic language which the computer can process, interpret and execute. This basic computer language is usually called assembly language, and it is composed of primitive orders to the computer. Examples of these are fetch a number from memory and place it in an accumulator or add the contents of one register to the contents of another one. Programming a computer in assembly language has always been a refined art usually reserved to the expert computer user.

As computers increased in size, speed and availability, the burden of programming in assembly language became excessive for many computer users. There was a need to communicate directions to the computer in simpler languages which described the computation to be performed in more natural terms. The computer user could use one language and the computer another language. The computer itself must execute the translation between these languages, and it does so by means of a translation program called a compiler. This concept and its effective implementation was another intellectual breakthrough in the development of computers. The language used by the computer user is called a higher level language, an application oriented language or a source programming language. Although there were simultaneous developments of this source language—compilation—assembly language idea, the first widely successful such development was the source language called FORTRAN and its associated compiler in the fifties. Since that period many other source languages have been developed such as ALGOL-60, COBOL, PASCAL, APL and ADA. The development of such programming systems has become as important as the development of computing machinery itself.

Programming languages have evolved greatly in the last thirty years providing ever increasing amounts of capability and congeniality to the programmer. From the point of view of scientific computation these languages have always been concerned with the expression of mathematical algorithms and the communication to the computer of the procedure for executing these algorithms. The original FORTRAN allowed the user to write mathematical formulas comprised of variables and the basic arithmetic operations, $+, -, \times, /$. In spite of the overwhelming advances made in the development of programming languages in the last thirty years, the capability of languages in the marketplace with respect to mathematical formulas has remained more or less the same. (There are of course notable exceptions such as APL and ALGOL-68 and ADA which provide facility for use of higher data types and corresponding operators.)

During the same thirty year period, the theory and practice of computer arithmetic has also had a significant development. The development of computer arithmetic as described in this article is composed of several levels of capability. These are the conventional capability, termed elementary computer arithmetic and three new capability levels termed basic, advanced and higher computer arithmetic. The computer programmer who executes algorithms exploiting the full range of this computer arithmetic must be able to conveniently write programs in terms of the operators and data types of these three new levels. Thus, we have seen the extension of certain scientifically oriented programming source languages, e.g., ALGOL [3], FORTRAN [5], [6], [7], PASCAL [8], to accommodate these capabilities. In this section we describe the way in which a source language is developed in order to properly interact with the three new levels of computer arithmetic, which we have discussed. We begin with the basic computer arithmetic framework.

6.1. Basic computer arithmetic. For this level of the source language we start with the type real (floating-point numbers). We extend the language from its customary setting of the four operations $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$ to the 15 fundamental operations

$$\begin{array}{ccccc}
\boxplus & \boxminus & \boxtimes & \boxslash & \boxdot \\
\triangledown\kern-0.65em{\scriptstyle +} & \triangledown\kern-0.65em{\scriptstyle -} & \triangledown\kern-0.65em{\scriptstyle \times} & \triangledown\kern-0.65em{\scriptstyle /} & \triangledown\kern-0.55em{\scriptstyle \cdot} \\
\vartriangle\kern-0.65em{\scriptstyle +} & \vartriangle\kern-0.65em{\scriptstyle -} & \vartriangle\kern-0.65em{\scriptstyle \times} & \vartriangle\kern-0.65em{\scriptstyle /} & \vartriangle\kern-0.55em{\scriptstyle \cdot}
\end{array}$$

While the three dot products $\boxdot$, $\triangledown\kern-0.55em{\scriptstyle \cdot}$, and $\vartriangle\kern-0.55em{\scriptstyle \cdot}$ operate on vectors, we avoid introducing a vector type at this level of the language. Instead, a new type called *dot precision* along with the associated procedure *dotadd* is introduced in terms of which the dot product operators $\boxdot$, $\overline{\nabla}$ and $\triangle\!\!\!\cdot$ may be composed. The following collection of constructs

dot precision,
:= (assignment from real to dot precision),
dotadd,
$\square, \nabla, \triangle$,

serve as primitives for developing the operations in the product spaces of the table of Fig. 6.

Apart from the assignment, the procedure dotadd and the roundings $\square, \nabla, \triangle$, no further operations, functions or procedures are required for this (auxiliary) type dot precision. Let $R = R(b, l, e1, e2)$ be the floating-point system of the computer in use. A variable of the type dot precision is a fixed-point variable with $d = t + 2e2 + 2l + 2|e1|$ digits of base b. See Fig. 10. For $n \leqq b^t$, every sum $\Sigma_{i=1}^{n} a_i \times b_i$ of floating-point products $a_i \times b_i$ can be represented as a variable of type dot precision. Moreover, every such sum can be computed in a local store of length d without loss of information (see Fig. 11).

A call for dotadd is given by

$$\text{dotadd}(A, b, c).$$

This makes the assignment[2]

$$A := A + b \times c,$$

where A is a variable of type dot precision and where $b \times c$ is the double length product of the variables b and c of type real. The addition indicated here is to be executed with complete accuracy. The exact inner product of two vectors $b = (b[i])$ and $c = (c[i])$ is now easily implemented with a variable a of the type dot precision and the procedure dotadd as follows.

```
$a := 0$;
**for** $i := 1$ **to** $n$ **do** dotadd $(a, b[i], c[i])$;
$x := a$;
```

The last statement $x := a$ rounds the value of the variable of type dot precision into the variable x of type R by applying the standard rounding $\square$ of the computer. x then has the value of the inner product $b \boxdot c$ which is within a single rounding error of the exact inner product $b \cdot c$. By changing the last statement $(x := a)$ in this program to

$$x := \text{realdown}\ (a),$$
$$\text{resp.}\quad x := \text{realup}\ (a),$$

the scalar products $b \overline{\nabla} c$ resp. $b \triangle\!\!\!\cdot\, c$ of the vectors $b = (b[i])$ and $c = (c[i])$ are produced.

For example, the method of defect correction requires highly accurate computation of expressions of the form

$$a \cdot b + c \cdot d,$$

with vectors a, b, c, d. Employing a variable x of type dot precision, this expression can

[2] In the remainder of this chapter we shall use a PASCAL-like representation of arithmetic operations. In particular, multiplication will be denoted by *. Exponentiation is denoted by the FORTRAN symbol **.

now be programmed as follows.

```
x := 0;
for i := 1 to n do dotadd (x,a[i],b[i]);
for i := 1 to n do dotadd (x,c[i],d[i]);
y := x;
```

This calculation is an example of our guiding principle. A result involving $2n$ multiplications and $2n-1$ additions is produced with but a single rounding operation.

6.2. Advanced computer arithmetic. Here the source language provides constructs to utilize the advanced computer arithmetic. We make use of a type concept and an operator concept as well as the overloading of (certain) function names. The type concept makes for easy use of the data types, VR, MR, IR, IVR, IMR, C, VC, MC, IC, IVC and IMC shown in Fig. 6. These many data types are implemented in terms of the four basic types: real, complex, (real) interval and complex interval and the two structurings: vector and matrix.

The operator concept makes for easy use of the many optimally accurate arithmetic operations associated with all these types. The operator concept required for these arithmetic purposes is limited, in principle, to an overloading feature. Types and operators for the sets mentioned above should be made available by the language in pre-defined and pre-compiled form. Thus

$$(a*x+b)*x+c$$

may be an expression for real or for complex matrices if the data and variables are correspondingly defined. Its value may be assigned to another variable of the resulting type by a single assignment:

$$y := (a*x+b)*x+c.$$

A side effect of such a short notation is that it obviously reveals its parallelism. Computation and assignment for all components of such a statement may be executed in parallel.

Traditional programming languages include standard functions such as sqrt, ln, sin and exp for the basic data type real. Such standard functions are now provided for each of the basic types real, complex, (real) interval and complex interval. As with all operations, this standard function capability is provided with maximal accuracy. One name is used for each standard function regardless of its argument type, that is, standard function names are overloaded.

6.3. Higher computer arithmetic. This level of source language deals with the capability of developing optimally accurate results for a class of numerical algorithms as described in §5. Computational techniques such as maximally accurate scalar products and defect correction methods were used to furnish this capability. Referring to that section, we select as an atom for this level of the source language, the capability to evaluate to maximum accuracy expressions composed of the data types and operations comprising the previous levels. In programming languages such expressions are often developed as program parts. Then this level of the source language extension is concerned with the specification that a program part shall be executed, equivalently, the corresponding expression be evaluated, with maximal accuracy.

We give a few examples of how such expression evaluations are encoded:

1. This example employs a new programming construct. Namely, a prefix *eval*, is

affixed to an expression. Thus, if a, c, e and f are vectors and B and D matrices, then the statement

$$f := \text{eval}(a + B * c + D * e)$$

delivers the value of $a + B * c + D * e$ to full accuracy. The assignment delivers the rounded value of the type of f. Possible assignments are as follows.

$Z :=$ rounds to an interval if Z is of type interval.
$z :=$ rounds to the nearest if z is of type R.
$z := <$ resp. $z := >$ rounds monotone downwardly resp. upwardly if z is of type R.

2. Another example is the computation of sums like

$$Z = \sum_{i=1}^{n} A_i B_i,$$

where the A_i, B_i, $i = 1(1)n$, are vectors or matrices. The corresponding source language encoding is

$$Z := \text{eval}\ (\text{sum}(A[i] * B[i], i = 1..n))$$

which delivers the value of Z, with the type of Z.

3. The new source language capability is substantive. Indeed the following examples demonstrate the encoding of expression evaluations which could not have been performed simply by applying optimal scalar products.

$$a := > \text{eval}(x + 4 * (3.0e7 * y/z)),$$
$$b := < \text{eval}(((4 * x - 5) * x + 3) * x + 25e3),$$
$$c := \text{eval}(\text{sum}(a[i] * x * * i, i = 1..n)).$$

The last computes the value of the polynomial

$$\sum_{i=0}^{n} a[i] x^i$$

with high accuracy.

In many cases it is more user-friendly to express the computation of expressions by means of a conventional program part. For example, suppose the expressions are already so encoded. The user desiring to upgrade results from such a piece of code so that they are highly accurate is not obliged to re-program. He may just upgrade his program.

For example, let PROG stand for the statement sequence of such a program part. Let x, y, z be names of those variables whose values are computed within PROG which are to be upgraded to outputs with high accuracy. This is accomplished in the following way

```
accurate x< ,y> , z do
    begin
      PROG
    end
```

This modified program computes x, y, z with high accuracy and rounds x downwardly, y upwardly, z to the nearest.

Needless to say, accurate evaluation of expressions or program parts is slower than execution with simple floating-point. However, accurate evaluation obviates the need for an error analysis. It also may be critical in unstable cases.

7. Final remarks. Here we comment on certain misconceptions about automatic computation and about certain deficiencies in practice which are not addressed by our approach. Then the current state of existing implementation of our approach is surveyed.

In colloquial use the terms precision and accuracy are synonymous. With respect to computers, *precision and accuracy represent quite different concepts*. It is surprising how much confusion this causes even among people who should know better. Precision refers to the quality of the tool whereas accuracy refers to the results produced by that tool. A computer may use great precision, i.e., its floating-point mantissa length l and its exponent range $e2-|e1|$ may both be quite large, but the same computer may be condemned to produce results of mediocre accuracy. The confusion between these two terms is very clearly revealed by computer manufacturers and those computer users who believe that more precision is an automatic ticket to higher accuracy in the result. Such a manufacturer offers a basic precision, a higher precision and perhaps even an extended precision. Such a user dissatisfied with the accuracy of the results obtained in single precision simply recomputes in a higher precision. We have already demonstrated by means of simple examples that this does not necessarily produce higher accuracy in the result. A mediocre crew of carpenters supplied with electron microscopes to mark their cuts will still produce a rickety house. We stress that the distinction between the concepts of precision and accuracy has always been maintained in this article.

When people talk numbers they talk decimal, even when conversing with their computer. Most computers then slyly talk another language to themselves (binary, octal, hexadecimal or whatever). This interface process is not an exact process. It is subject to rounding errors with an associated loss of information. This places a burden on the user which is frequently not just a minor annoyance. Certainly financial calculations, among others, are affected by this interface problem.

The use of nondecimal number systems in the computer is a historical development stemming from considerations ultimately based on cost and performance. Modern technology eliminates this need to deal with nondecimal number systems. The user should not be burdened by the interface conversion problem. If computers also talk decimal to themselves, they will be more user-friendly.

Most computer users have experienced changing computers. With varying degrees of trauma, they have learned to deal with the idiosynchrocies of the new system. Standardization of computer systems is certainly an ideal which is a long way off. However, standardization of the computer arithmetic is at hand. The theory and practice of computer arithmetic as discussed in this article provide an excellent vehicle for this standardization. The methodology is well founded in fundamental mathematical principles, and the implementation techniques are efficient and practical as well. The results are of maximal accuracy, and the procedures are user-friendly.

It is of interest to survey the implementations of the methodology discussed in this article which are already in existence: Basic arithmetic is available in a software implementation in all IBM System 370 computers. Based on this, many parts of advanced and of higher order computer arithmetic routines are available in the form of subroutines and program packages. A commercially available such package is called ACRITH [13], [14]. One particular IBM mainframe, the 4361, offers all basic arithmetic capabilities in hardware supported microcode.

Basic, advanced and higher order arithmetic routines, embedded in a PASCAL-SC environment, are available in two micros, the ZILOG Z-80 and the MOTOROLA-68000.

A hardware unit which performs all basic arithmetic routines has been built in bit slice technology. It may be used as an arithmetic unit in connection with micros or mainframes.

Basic and advanced computer arithmetic has been embedded into FORTRAN [5] and FORTRAN 8X [6], [7]. The programming language Matrix-PASCAL [8] supports basic, advanced and higher arithmetic.

Recall that basic arithmetic comprises the 15 operations enumerated in §4. The arithmetic of the so-called IEEE standard provides for 12 of these 15 operations. These 12 operations are available on the INTEL 8087 chip, among others [10], [15]. We recommend that future versions of such chips implement the optimal scalar product so that all 15 of the operations of basic arithmetic are available. At the very least the full double length product should be available for all precisions; this capability providing a basis for efficient simulation of optimal scalar products.

REFERENCES

[1] G. ALEFELD AND J. HERZBERGER, *Einführung in die Intervallrechnung*, Reihe Informatik, 12, Bibliographisches Institut Mannheim, 1974.

[2] ______, *Introduction to Interval Computations*, Academic Press, New York, 1983.

[3] N. APOSTOLATOS et al., *The algorithmic language Triplex*-ALGOL-60, Numer. Math., 11 (1968), pp. 175–180.

[4] H. BÖHM, *Evaluation of arithmetic expressions with maximum accuracy*, in [20], pp. 121–137.

[5] G. BOHLENDER, et al., FORTRAN *for contemporary numerical computation*, Computing, 26 (1981), pp. 277–313.

[6] ______, *Proposal for arithmetic specification in* FORTRAN 8X, Institute for Applied Mathematics, Univ. Karlsruhe, 1982.

[7] ______, *Application module: scientific-computation for* FORTRAN 8X, Institute for Applied Mathematics, Univ. Karlsruhe, 1983.

[8] ______, *Matrix* PASCAL, in [20], pp. 311–384.

[9] G. BOHLENDER AND U. KULISCH, *Features of a hardware implementation of an optimal arithmetic*, in [20], pp. 269–290.

[10] J. COONAN, et al., *A proposed standard for floating-point arithmetic*, SIGNUM Newsletter, 1979.

[11] J. J. DONGARRA, J. R. BUNCH, C. B. MOLER AND G. W. STEWART , LINPACK *Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[12] G. E. FORSYTHE, *Pitfalls in computation, or why a math book isn't enough*, Computer Science Department, Stanford Univ., Stanford, CA, 1970.

[13] *High-Accuracy Arithmetic, Subroutine Library*, General Information Manual, IBM Program Number 5664–185, 1984.

[14] *High Accuracy Arithmetic, Subroutine Library*, Program Description and User's Guide, IBM Program Number 5664–185, 1984.

[15] INTEL 12 1586-001, *The* 8086 *family user's manual*, Numeric Supplement, 1980.

[16] E. W. KAUCHER AND W. L. MIRANKER, *Self-Validating Numerics for Function Space Problems, Computation with Guarantees for Differential and Integral Equations*, Academic Press, New York, 1984.

[17] E. KAUCHER AND S. M. RUMP, *Generalized iteration methods for bounds of the solution of fixed point operator-equations*, Computing, 24 (1980), pp. 131–137.

[18] U. KULISCH, *Grundlagen des Numerischen Rechens-Mathematische Begründing der Rechnerarithmetik*, Bibliographisches Institut, Mannheim, 1976.

[19] U. W. KULISCH AND W. L. MIRANKER, *Computer Arithmetic in Theory and Practice*, Academic Press, New York, 1981.

[20] ______, *A New Approach to Scientific Computation*, Academic Press, New York, 1983.

[21] U. KULISCH AND CH. ULLRICH, *Wissenschaftliches Rechnen und Programmiersprachen*, Berichte des German Chapter of the ACM 10, Teubner Verlag, 1982.

[22] M. A. MALCOLM, *On accurate floating-point summation*, Comm. ACM, 14 (1971) pp. 731–736.

[23] R. E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1966.
[24] ______, *Methods and Applications of Interval Analysis*, SIAM Studies in Applied Mathematics 2, Society for Industrial and Applied Mathematics, Philadelphia, 1979.
[25] S. M. Rump, *How reliable are results of computers*, Jahrbuch Überblicke Mathematik, 1983, Bibliographisches Institut, Mannheim, 1983, pp. 163–168.
[26] ______, *Solving algebraic problems with high accuracy*, in [20], pp. 51–120.
[27] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford Univ. Press, Cambridge, 1965.
[28] ______, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1963.
[29] J. M. Yohe, *Roundings in floating-point arithmetic*, IEEE Trans. Comput. C-12 (1973), pp. 577–586.

SUPPLEMENTARY BIBLIOGRAPHY

[S1] J. G. P. Barnes, *An overview of Ada*, Software Practice and Experience 10, 11 (1980), pp. 851–887.
[S2] A. K. Cline, C. B. Moler, G. W. Stewart and J. H. Wilkinson, *An estimate for the condition number of a matrix*, SIAM J. Numer. Anal., 16 (1979), pp. 368–375.
[S3] W. Cody and W. White, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
[S4] J. D. Dixon, *Estimating extremal eigenvalues and conditions numbers of matrices*, SIAM J. Numer. Anal., 20 (1983), pp. 812–814.
[S5] M. D. Ercegovac, *A general hardware-oriented method for evaluation of functions and computations in a digital computer*, IEEE Trans. Comput., C-26 (1977), pp. 667–680.
[S6] W. W. Hager, *Condition estimates*, SIAM J. Sci. Stat. Comp., 5 (1984), pp. 311–316.
[S7] D. Kuck, *The Structure of Computers and Computations*, John Wiley, New York, 1978.
[S8] MACSYMA *Reference Manual*, Mathlab Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, 1983.
[S9] D. P. O'Leary, *Estimating matrix condition numbers*, SIAM J. Sci. Stat. Comp., 1 (1980), pp. 205–209.
[S10] F. N. Ris, *Interval analysis and applications to linear algebra*, Dissertation, Oxford Univ., 1972.
[S11] E. Swartzlander, *Computer arithmetic*, Benchmark Papers in Engineering and Computer Science 21, E. Swartzlander, ed., Hutchinson Ross, 1979.
[S12] E. Wegbreit, *The treatment of data types in* ELI, Comm. ACM, 17 (1974), pp. 251–264.