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 $\boxdot$, $\bigtriangledown$, $\bigtriangleup$ 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.

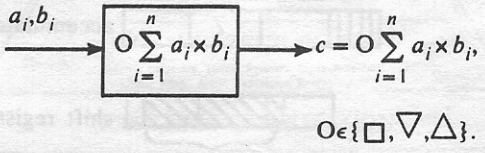


FIG. 9. *The black box for scalar products.*

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

$$\bigcirc \sum_{i=1}^{n} c_i, \qquad \bigcirc \in \{\boxdot, \bigtriangledown, \bigtriangleup\}$$

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.

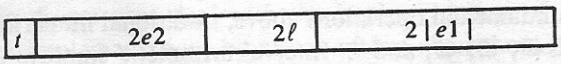


FIG. 10. *Register for scalar product accumulation.*

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.

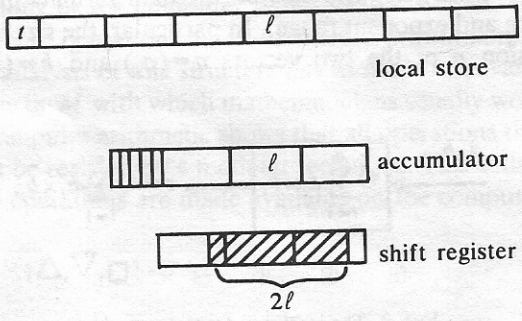


FIG. 11. *Addition process for scalar products.*

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.

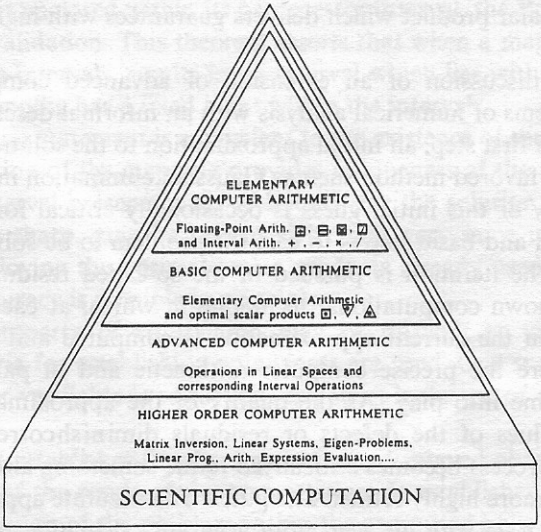
Other addition techniques or carry handling processes are possible. While the addition process described was in terms of hardware registers, it can, of course, also be simulated in software. For a more detailed discussion of these principles, see [8] [9], [20]. Special purpose long accumulators have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



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.