Comput. Math. Applic. Vol. 14, No. 1, pp. 53–69, 1987
Printed in Great Britain.

0097-4943/87 $3.00 + 0.00

SCIENTIFIC COMPUTING

AN INTRODUCTION TO THE SCIENTIFIC COMPUTING LANGUAGE Pascal-SC

L. B. RALL
Mathematics Research Center, University of Wisconsin–Madison, Madison, WI 53706, U.S.A.

Abstract—Microcomputers are now widely used for small-scale scientific, engineering, and statistical computing. Pascal-SC (Pascal for Scientific Computing) is a language developed specifically for this purpose. Its most important features are: (i) accurate floating-point arithmetic for real, complex, and interval numbers, vectors, and matrices, with controlled rounding if desired; (ii) the convenience of operator notation for numerical data types and the ability to accept user-defined operators for nonstandard data types to make programs easier to write, read, and document; and (iii) compatibility with ordinary Pascal, so that Pascal programming techniques and programs already written in Pascal can be used immediately. In Pascal-SC, solutions of linear systems of equations, inverses of matrices, and eigenvalues and eigenvectors are computed with guaranteed error bounds, and scalar products of vectors and sums of arbitrary length of floating-point numbers are computed to the closest floating-point number, or rounded upward or downward as desired. These basic features of Pascal-SC will be described, together with some applications.

OBJECTIVE

A brief description of the language Pascal-SC (Pascal for Scientific Computation) will be given to explain features of this extension of ordinary Pascal [1] which are particularly useful for scientific, engineering, and statistical calculations on microcomputers. The reader is assumed to be familiar with Pascal programming on at least an introductory level, and to have some experience with numerical computation in scientific and engineering problems. With this background, it should be easy to appreciate the advantages of Pascal-SC.

In order to keep the discussion short and to the point, many details will be omitted, and a formal description of the language will not be given. For operational and programming details, the reader should consult Refs [2–4]; formalities are given in Ref. [5]. Here, simple examples will be used to illustrate ideas as they are introduced.

WHY Pascal-SC?

As pointed out by Wirth [1], the introduction of a new computer language requires careful justification. The same applies to an extension or modification of an existing language, particularly a language which is as successful and widely used as Pascal. The most important additional features of Pascal-SC are:

(i) accurate, trustworthy floating-point arithmetic with controllable rounding;
(ii) user-defined operators to facilitate programming and documentation:
furthermore, it is of considerable importance to note that:
(iii) Pascal-SC retains the features of ordinary Pascal.

Thus, none of the investment in learning to program in Pascal or in programs already written in Pascal is lost in going from Pascal to Pascal-SC. The Pascal-SC compiler will translate programs written in ordinary Pascal.

With regard to (i), the floating-point arithmetic provided by Pascal-SC is implemented not only for real floating-point numbers (type REAL), but also for complex numbers, intervals, and vectors and matrices of these types. This allows convenient and accurate computation with the kinds of numerical data most frequently encountered in scientific and engineering problems. The floating-point arithmetic of Pascal-SC [4] is constructed on the basis of the theory of computer

arithmetic given by Kulisch and Miranker [6], guaranteeing accuracy, controllability and reliability of the results.

The second important additional capability of Pascal-SC is that it allows definition of operators to manipulate numerical data types in ordinary mathematical notation. For example, if A is a matrix, and x, b, c are vectors, then the programmer can write the statement

$$c: = A * x + b; \tag{1}$$

in Pascal-SC to perform the indicated calculations. This notation follows ordinary mathematical usage, and thus has the advantages of clarity and simplicity compared to the calling of procedures and functions in ordinary Pascal. In order for the Pascal-SC compiler to accept (1), the heading of the program has to contain a definition of the binary operators * to perform matrix by vector multiplication and + to perform vector addition. (Source code for these operators is included in the Pascal-SC package for vector and matrix arithmetic.) In addition to this "overloading" of standard operator symbols, Pascal-SC permits the user to give operators arbitrary names (e.g. XOR for "exclusive or"), and assign priorities to such operators. One particular convenience of Pascal-SC is that the operator ** can be defined to perform exponentiation, which makes the writing of polynomials and other functions containing powers simpler than in ordinary Pascal. In allowing user-defined operators, Pascal-SC is similar to Algol 68 and Ada.

These points will be discussed in more detail in the following sections.

FLOATING-POINT REAL ARITHMETIC

This is the "built-in" arithmetic of Pascal-SC for floating-point numbers (type REAL), and is accurate, controllable, and reliable. Before going on to details, a precise statement of the meaning of these terms is necessary because the related concepts of "accuracy" (the exactness with which the results are calculated) and "precision" (the number of digits used in the representation of floating-point numbers) are often confused. For example, the result 32.0–31.0 = 1.00 is calculated with low precision (3 decimal digits), but high accuracy (exactly). By contrast, UNIVAC 1100 floating-point hardware gives

$$134217728.0 - 134217727.0 = 2.00000000,$$

which is done with higher precision (9 significant digits), but no accuracy, since the correct answer is 1.00000000 [7].

It is possible to discuss the idea of accuracy independently of the particular precision used, since each floating-point system contains only a finite set of numbers. In a given system S, two floating-point numbers u, v with $u < v$ will be said to be *adjacent* if there is no floating-point number w such that $u < w < v$. For $x, y \in S$, the exact result of $x \circ y$ of an arithmetic operation $\circ$, where $\circ \in \{+, -, *, /\}$, will either be a floating-point number, or a real number w such that $u < w < v$, where u, v are adjacent floating-point numbers. In order to produce a floating-point number in the latter case, w is "rounded" to an element of S, which should be either u or v. This is the basic requirement for *reliability* of a floating-point arithmetic operation. The floating-point software supplied for some microcomputers does not meet this simple requirement, and neither does the floating-point hardware of some mainframe computers. This unhealthy situation has lead the IEEE to undertake the promulgation of standards for floating-point arithmetic (see the *SIGNUM Newsletter* for October, 1979).

An *accurate* rounding R of the floating-point operation $\circ$ selects $R(x \circ y)$ equal to u or v to minimize the roundoff error

$$e(x \circ y) = |R(x \circ y) - (x \circ y)|. \tag{2}$$

This is the best possible answer (BPA) rounding [8]. In case of a tie between u and v, a suitable rule is invoked. In Pascal-SC, this rounding is to the one of u, v which is furthest from zero, in order to satisfy the condition of antisymmetry of $R, R(-x) = -R(x)$, required by the general theory [6].

The distance $|u - v|$ between u and v will depend on the precision of the floating-point numbers being used; this determines the maximum roundoff error of a reliable calculation. Of course, if the

arithmetic of the machine being used is not reliable, then roundoff error is not related in a simple way to precision, and the attempt to "buy" more accuracy by using increased precision can be futile.

There are other ways in which rounding to a reliable result can be carried out, including:

(i) w is rounded upward to v;
(ii) w is rounded downward to u.

The directed roundings (i) and (ii) are necessary to support interval arithmetic [6, 9, 10]. Rounding in a floating-point arithmetic is said to be *controlled* if the user can choose the method desired for the result of a given floating-point arithmetic operation Pascal-SC gives the user the choice of BPA and the upward and downward directed roundings, which means that a total of *twelve* operators are provided for the four basic arithmetic operations $+/, -, *, /$, and the three roundings listed above:

$$\begin{cases} +, -, *, /, & \text{BPA rounding} \\ +>, ->, *>, />, & \text{Upward rounding} \\ +<, -<, *<, /<, & \text{Downward rounding.} \end{cases} \tag{3}$$

These operations are also required by the IEEE standard. Thus, the Pascal-SC programmer can control the direction of rounding, for example, to obtain guaranteed lower or upper bounds for the values of arithmetic expressions [11]. It also follows from the reliability of Pascal-SC arithmetic that the addition and multiplication operators (3) are commutative, which is not true for the kind of floating-point arithmetic ordinarily encountered.

For the microcomputer implementations of Pascal-SC, decimal arithmetic is used, and the precision of floating-point numbers is 12 decimal digits in scientific notation, with an exponent range in powers of 10 from -99 to $+99$. The smallest and largest positive numbers are thus MINREAL = 1.00000000000E − 99 ($= 10^{-99}$) and MAXREAL = 9.99999999999E + 99, respectively. Zero is represented by 0 = 0.00000000000E + 00, as usual [4]. The use of decimal arithmetic avoids the errors introduced by conversion between binary and decimal upon input and output. Decimal values are represented internally by two BCD digits per byte. The precision of the Pascal-SC floating-point number system for microcomputers is thus adequate for the representation of most numerical quantities of interest in scientific and engineering computation. Furthermore, since Pascal-SC floating-point arithmetic is reliable, accurate, and controllable, there is seldom any need for more than 12 decimal digits of precision in a given computation.

To illustrate these features of Pascal-SC arithmetic, consider the product

$$A = (13.4565432278)(0.000453782392145). \tag{4}$$

The exact value of A is not a 12-digit floating-point number, so rounding will take place in the corresponding floating-point multiplication operations. The floating-point operation * gives

$$B = (1.34565432278\text{E} + 01) * (4.53782392145\text{E} - 04) = 6.10634237591\text{E} - 03, \tag{5}$$

as the BPA for A, which is in error by at most $\delta/2 = 5.0 \times 10^{-15}$, since Pascal-SC arithmetic is reliable and the distance between B and its two neighboring floating-point numbers is $\delta = 10^{-14}$. More precisely, (5) establishes that A belongs to the half-open interval $[B - \delta/2, B + \delta/2)$, since ties are rounded away from 0. The operations $*<$ and $*>$ with directed rounding give

$$\begin{aligned} C &= (1.34565432278\text{E} + 01) *< (4.53782392145\text{E} - 04) = 6.10634237591\text{E} - 03, \\ D &= (1.34565432278\text{E} + 01) *> (4.53782392145\text{E} - 04) = 6.10634237592\text{E} - 03, \end{aligned} \tag{6}$$

respectively. Since $C = B$, the result (6) shows that the exact answer A belongs to the interval $[C, D] = [B, D] = [B, B + \delta]$; therefore, on the basis of (5), A belongs to the half-open interval $[B - \delta/2, B + \delta/2) \cap [B, B + \delta] = [B, B + \delta/2)$. Thus, this calculation proves that $A \geqslant B$ and $A - B < 5.0 \times 10^{-15}$. This gives a more accurate location for A than the BPA answer (3.4) by itself.

In addition to the twelve rounded arithmetic operations listed above, Pascal-SC provides the most frequently used standard functions, computed to BPA accuracy [4]. In order to keep the microcomputer version of the Pascal-SC system small, extensive use is made of external libraries,

so the compiler will bring in code only for functions, procedures, and operators actually needed by the program.

FLOATING-POINT COMPLEX ARITHMETIC

In Pascal-SC, manipulation of complex numbers is accomplished by pretranslated subroutines for operators, functions, and procedures. For this reason, the declaration of complex numbers has the stereotyped form

```
TYPE COMPLEX = RECORD RE, IM: REAL END;
```

Thus, the representation of a complex number Z in Pascal-SC is in Cartesian coordinates. For input and output of Z, the standard format is (Z.RE, Z.IM).

In ordinary Pascal, addition and other arithmetic operations with complex numbers have to be done by procedures and functions. In Pascal-SC, however, operator overloading simplifies notation in the program considerably. To divide the complex number V by the complex number W and assign the result to the complex number Z, one writes only

```
Z: = V/W;                                                    (7)
```

in the program. Arithmetic operators are defined for all pairs of operands of types INTEGER, REAL, and COMPLEX, since these occur naturally in expressions being evaluated. If K, R, C denote generic variables of types INTEGER, REAL, and COMPLEX, respectively, then six addition operators are needed:

```
+C, K + C, C + K, R + C, C + R, C + C.                       (8)
```

Similarly, six subtraction operators

```
-C, K - C, C - K, R - C, C - R, C - C,                       (9)
```

are required, as well as five multiplication and five division operators:

```
K*C, C*K, R*C, C*R, C*C,
K/C, C/K, R/C, C/R, C/C.                                     (10)
```

All 22 of the operators (8)–(10) are provided in an external library in the form of pretranslated code, and the corresponding declarations are available to the programmer in an external text file. Actual coding is also simplified considerably the fact that the programmer can direct the compiler to refer to external libraries for type declarations and definitions of operators and other needed functions and procedures by a statement of the form $USES COMPLEX; [3, 4]. This feature makes the source code of a Pascal-SC program more compact and readable.

All complex floating-point operations in Pascal-SC calculate the real and imaginary parts of the result to BPA accuracy. Multiplication and division require special algorithms to attain this accuracy [4, 6]. For example, consider a function CDIV which does complex division by the usual formula:

```
FUNCTION CDIV(A, B: COMPLEX): COMPLEX;
  VAR DENOM: REAL;
      U: COMPLEX;
  BEGIN
      DENOM := B.RE*B.RE + B.IM*IM;
      U.RE := (A.RE*B.RE + A.IM*B.IM)/DENOM;
      U.IM := (A.IM*B.RE - A.RE*B.IM)/DENOM;
      CDIV := U
  END;                                                       (11)
```

The Pascal-SC operator for complex division,

```
OPERATOR/(A,B: COMPLEX) RES: COMPLEX
  EXTERNAL 182;                                              (12)
```

has a number of advantages over the function CDIV. Among these are:

Accuracy

Ordinary Pascal functions and procedures for COMPLEX multiplication or division, such as CDIV given above, cannot attain BPA accuracy because of roundoff errors. Furthermore, catastrophic cancellations can happen which render the results almost meaningless when the ordinary formulas in (11) are used. Even in less drastic situations, a number of significant digits can be lost without warning if (11) is used. For example,

$$\begin{aligned} V &= (1.23456789, 1.23456789), \\ W &= (1.0000123E-05, 1.0000321E-05), \end{aligned} \tag{13}$$

one gets

$$\begin{aligned} V/W &= (1.23454048308E+05, -1.22216794612E+00), \\ CDIV(V, W) &= (1.23454048308E+05, -1.22216\mathbf{773505}E+00), \end{aligned} \tag{14}$$

where the incorrect digits of CDIV(V, W) are indicated in **boldface**. Even in this fairly harmless-looking case, the algorithm (11) has lost five significant digits in the imaginary part of the quotient in a single division, while all the digits of the Pascal-SC result for V/W are correct. To be sure, roundoff error will also increase with repeated use of the Pascal-SC division operator (12), but at a slower and more predictable rate.

Deferred overflow

In the Pascal-SC algorithms for complex multiplication and division, overflow does not occur unless the real or imaginary part of the *result* is actually > MAXREAL in absolute value, whereas overflow can occur in (12) in the calculation of the intermediate values DENOM, U.RE, U.IM, even though the final result is representable. For example, for

$$V = (3.0E+99, -1.0E+99), \quad W = (1.0E+99, -1.0E+99), \tag{15}$$

Pascal-SC complex division gives the result

$$V/W = (2.00000000000E+00, 1.00000000000E+00), \tag{16}$$

while the subroutine (12) for CDIV overflows when trying to compute DENOM.

Ease of use

For VAR U, V, W: COMPLEX, the use of (12) allows one to write

```
U := V/W;
```
(17)

in the source code for the program instead of

```
U := CDIV(V, W);
```
(18)

as in ordinary Pascal. In the case of complicated expressions involving complex numbers, the gain in programming ease using ordinary mathematical notation with operators instead of function and procedure calls is significant. The source code is more likely to be correct in the first place, and also will be easier to document and read later.

Compilation time

The function CDIV has to be compiled from the source code (11) for each ordinary Pascal program which uses complex division. The operator (12), on the other hand, is given by pretranslated code which is automatically linked to the user's program in the last stage of the compilation. This saves compilation time.

The accurate complex division in Pascal-SC turns out to be slower than the function CDIV. According to the table given in [12, p. 267], a typical time for the complex division V/W is 100 ms for a 2.5 MHz Z80 processor. The function CDIV(V/W) uses six real multiplications, two real divisions and three real addition/subtractions. The typical times for these operations given in Ref. [12] total 60.4 ms. It will be seen later that some Pascal-SC operations are actually faster than

their inaccurate real simulations; however, in the case of complex multiplication and division, one pays a little for guaranteed, reliable accuracy.

In addition to the arithmetic operators $+, -, *, /$ for type COMPLEX, a number of additional operators, functions, and procedures are provided in the Pascal-SC complex library for convenience. For details on these, including the domains and ranges of the standard functions, see Ref. [4].

Rounded complex operations $+<, +>, -<, ->, *<, *>, /<, />$ are also included in an external library in the form of pretranslated code [4]. Here, rounding is carried out componentwise. Each complex number $z = (x, y)$ with $|x|, |y| <$ MAXREAL will belong to a rectangle with corners which are the floating-point complex numbers $A = (u, v)$, $B - (u + \delta, v)$, $C = (u + \delta, v + \eta)$, $D = (u, v + \eta)$, and which contains no other floating-point complex numbers. The result of rounding z downward will be $\nabla z = A = (u, v)$, while z is rounded upward to $\Delta z = C = (u + \delta, v + \eta)$, where δ and η are the spacings in the floating-point screen in the horizontal and vertical directions in the complex plane, respectively. The BPA rounding of z will be (BPA(x), BPA(y)), and thus could be any one of the four points A, B, C, D.

Complex arithmetic with directed rounding can be used as the basis for complex interval arithmetic [6].

FLOATING-POINT INTERVAL ARITHMETIC

Interval arithmetic [9, 10, 13] is based on the use of closed, finite intervals [a, b] of real numbers as its basic elements. Interval arithmetic has a number of significant applications in scientific, engineering, and statistical computation; however, its use has not been widespread up to now because of the limitations of conventional computer arithmetic units [14] and ordinary programming languages. In Pascal-SC, interval arithmetic has been implemented efficiently, and is just as convenient to use as real or complex arithmetic.

In the general theory of computer arithmetic [6], interval arithmetic is regarded as a special case of arithmetic on subsets of real numbers. Here, if X, Y are subsets of $\mathbf{R}$, and $\circ \in \{+, -, *, /\}$, then

$$Z = X \circ Y = \{x \circ y \mid x \in X, y \in Y\}, \tag{19}$$

by definition. For division, of course, $0 \in Y$ is excluded. If $X = [a, b]$ and $Y = [c, d]$ are intervals, then $Z = [r, s]$ is an interval if defined, and the endpoints r, s of Z can be calculated from the endpoints of X, Y [6, 9, 10]. It is assumed, of course, that the intervals $X = [a, b]$ considered are finite and *proper*, i.e. $a \leqslant b$.

The most basic application of interval arithmetic is the following: if the value of a function, for example,

$$w = 9x^4 - y^4 + 2y^2, \tag{20}$$

is calculated in interval arithmetic for intervals X, Y, then the resulting interval W will contain all values w of the function (20) for all values of $x \in X, y \in Y$. This property of interval arithmetic allows one to bound the ranges of functions without detailed analysis of maxima and minima. Since rounding of interval operations is outward to the smallest floating-point interval which contains the exact results [4], automatic bounds for round-off error can be obtained conveniently by taking the input intervals X, Y to be single points, that is, $X = [x, x]$, $Y = [y, y]$ [9, 10, 13]. For a simple application of this principle, consider the evaluation of (20) by the expressions

$$w := 9*(x**4) - y**4 + 2*(y**2); \tag{21a}$$

$$w := (3*(x**2) - y**2)*(3*(x**2) + y**2) + 2*(y**2); \tag{21b}$$

in REAL arithmetic, and

$$W := 9*(X**4) - Y**4 + 2*(Y**2); \tag{22a}$$

$$W := (3*(X**2) - Y**2)*(3*(X**2) + Y**2) + 2*(Y**2); \tag{22b}$$

in INTERVAL arithmetic, respectively. The power operators ** in (21) and (22) used to obtain the following results are given by (37) and (38), respectively. The actual Pascal-SC program used is given in the Appendix.

For $x = 10864, y = 18817$, the statements (21) and (22) give

$$w = 1.58978 \times 10^5,$$

$$W = [-8.410220 \times 10^5, 1.158978 \times 10^6], \tag{23a}$$

$$w = 1.00000000000,$$

$$W = [1.00000000000, 1.00000000000]. \tag{23b}$$

The enormous width of W in (23a) indicates that the values given for w can be subject to large roundoff error, and hence are untrustworthy. On the other hand, evaluation of (20) by the statement (21b) shows that this formulation is highly accurate, and in fact proves that (21a) yields the exact value $w = 1$ of the function (20) for the given values of x and y. This example also shows that the way in which arithmetic expressions are written can be crucial for accuracy. The techniques for evaluation of arithmetic expressions with maximum accuracy or directed rounding [11, 15] will be incorporated as a standard feature in a forthcoming version of Pascal-SC.

In Pascal-SC, floating-point intervals are declared in the stereotyped way:

```
TYPE INTERVAL = RECORD INF,SUP: REAL END;
```

In order to preserve the inclusion property of interval arithmetic, all rounding in floating-point INTERVAL arithmetic is *outward*: The result given for $X \circ Y$ is the smallest floating-point interval Z which contains the actual result.

Mixed arithmetic is defined only between types INTERVAL and INTEGER, because REAL floating-point expressions do not necessarily yield the exact values of their results, and hence the resulting interval might not contain the true outcome of the computation. In addition to interval versions of standard functions and various utility procedures [4], the interval library includes some operators which work with intervals as sets of real numbers. There are the "lattice operators"

$$\begin{cases} ** \text{ Intersection}, \\ ++ \text{ Interval Hull}, \end{cases} \tag{24}$$

and the relational operators

$$\begin{cases} <= \text{ Subinterval}, \\ >= \text{ Superinterval}, \\ >< \text{ Disjointness}, \\ \text{IN Point Inclusion}, \end{cases} \tag{25}$$

[4]. The intersection operator ** will generate an error interrupt if its operands are disjoint intervals, otherwise, their intersection $I \cap J$ will be computed. If $I = [a, b]$ and $J = [c, d]$, then $I ++ J = [\min\{a, c\}, \max\{b, d\}]$ is the smallest interval which contains both I and J. The relation $I \leqslant J$ is TRUE if $I \subseteq J$, otherwise FALSE; similarly, $I \geqslant J$ is TRUE if $I \supseteq J$. The result of $I \gtrless J$ is TRUE if I and J are disjoint intervals. This test can be used to avoid $I ** J$ in this case. If R is a floating-point number (type REAL), then R IN I is TRUE if $R \in I$ as a real number.

The standard function ISQR is provided to square intervals. Its use gives better results in general than multiplication because, for example, $[-1, 1]^2 = [0, 1]$, while $[-1, 1] * [-1, 1] = [-1, 1]$.

With regard to the efficiency of the implementation of interval arithmetic in Pascal-SC, Bohlender and Grüner [12] give the following typical times in milliseconds for a microcomputer with a 2.5 MHz Z80 processor:

Operation	+	−	*	/
REAL	2.2	2.2	6.0	10.0
INTERVAL	5.4	5.4	23.0	31.0
COMPLEX	5.4	5.4	45.0	100.0

Considering the fact that each interval operation has to determine two real numbers, and that interval multiplication and division require the calculation of four real products in one case [6], the above indicates an almost optimal implementation in software. By contrast, factors of 100 or 200 between REAL and INTERVAL arithmetic speed have been noted on conventional comput-

ers, such as the UNIVAC 1100 series [14]. In addition, while a REAL computation provides only a floating-point number which approximates the result, an INTERVAL computation on the other hand provides an interval which is guaranteed to contain the true result. An important application of this property is to the solution of linear systems of equations with guaranteed error bounds. It should be noted that INTERVAL multiplication and division are faster than the corresponding COMPLEX operations, as is to be expected.

REAL FLOATING-POINT VECTOR AND MATRIX ARITHMETIC

Calculations with vectors and matrices are among the most commonly encountered tasks in scientific, engineering, and statistical computation. Pascal-SC offers the user the same reliability, accuracy, controllability, and convenience when calculating with real n-dimensional floating-point vectors and matrices as it does for the scalar types REAL, COMPLEX, and INTERVAL. This is accomplished with the aid of an external source code library of operators, functions, and procedures for vector and matrix manipulation, and the built-in function SCALP for the calculation of scalar products of floating-point vectors to BPA accuracy or with directed rounding at the option of the user.

Convenience

In order to illustrate the convenience of Pascal-SC for vector and matrix calculations, suppose that A, B are $n \times n$ matrices, and x, y, z are n-dimensional vectors. To evaluate

$$z = 5.5ABx + 3y, \tag{26}$$

the corresponding expression is Pascal-SC is

$$\mathrm{z} := 5.5 * \mathrm{A} * \mathrm{B} * \mathrm{x} + 3 * \mathrm{y}; \tag{27}$$

which uses ordinary operator notation instead of the function and procedure calls which would be required in ordinary Pascal and most other languages. In order to make use of the software provided in the corresponding external library, a stereotyped declaration of floating-point vector and matrix data types is expected:

```
CONST DIM = n; {The actual dimension replaces n}
TYPE  DIMTYPE = 1..DIM;
      RVECTOR = ARRAY [DIMTYPE] OF REAL;
      RMATRIX = ARRAY [DIMTYPE] OF RVECTOR;
```

The operators $+, -, *$, and the operators $+<, +>, -<, ->, *<, *>$ with directed rounding are available for various permissible combinations of operands, for example, multiplication of an RVECTOR by an INTEGER or REAL, and so on [4]. A forthcoming version of Pascal-SC will allow vectors and matrices to be dimensioned dynamically when the program is executed, thus eliminating one of the annoying restrictions of ordinary Pascal.

Reliability, accuracy, and controllability: the scalar product SCALP

The general theory of computer arithmetic [6] requires that each component of the result of a vector or matrix operation be rounded to the BPA for the actual real result, or downward or upward to an adjacent floating-point number if desired. Addition and subtraction of vectors and matrices present no problems from the standpoint of this requirement, since the desired results can be calculated componentwise with the aid of the six REAL arithmetic operators $\pm, \pm<, \pm>$ described earlier. Calculation of the scalar products of vectors, which is an inherent component of matrix and matrix by vector multiplication, is a different matter. Ordinarily, this calculation is simulated by a FOR loop of real operations, such as in the following function:

```
FUNCTION SPROD(A,B: RVECTOR): REAL;
  VAR I: DIMTYPE; S: REAL;
  BEGIN
    S := 0;
```

```
    FOR I := 1 TO DIM DO
      S := S + A[I] * B[I];
    SPROD := S
  END;                                                    (28)
```

This is an example of what Kulisch calls the "vertical" definition of computer arithmetic [6, 16]. Of course, there is no hope that the result of SPROD will be accurate in general. For this reason, the internal calculations in a function of this kind are often done in higher precision than the external calculation. While this is a often a great help in some cases, it still does not *solve* the accuracy problem. On the other hand, the Pascal-SC function

```
SCALP(A, B: RVECTOR; ROUND: INTEGER);                     (29)
```

a built-in feature of the compiler, will round the *exact* value of the scalar product of A and B to an *adjacent* floating-point number, with rounding downward, upward, or to the BPA controlled by the value of the parameter ROUND [4]. This reliability, accuracy, and controllability can be achieved by special algorithms [6], or the provision of a sufficiently long accumulator. In the microcomputer version of Pascal-SC, this "long accumulator" is implemented in software [12], but the same thing can be done in hardware [17], and is expected to be a feature of future advanced mainframe computers.

In order to allow the accumulation of several scalar products, the parameter ROUND can also inhibit the clearing of the long accumulator before the product is calculated [4]. The corresponding values are given in the following table:

Rounding	Clear long accumulator	Inhibit clearing
BPA	ROUND = 0	ROUND = 4
Downward	ROUND = −1	ROUND = 3
Upward	ROUND = 1	ROUND = 5

If the long accumulator is not being cleared, other arithmetic operations are not permitted between successive calls of SCALP [4].

The use of SCALP makes it possible to calculate the results of matrix and matrix by vector multiplications to the closest floating-point numbers, or rounded to the closest larger or smaller neighboring values if desired. This distinguishes Pascal-SC vector and matrix arithmetic from traditional packages.

Some of the important properties of SCALP are illustrated by the following examples.

Accuracy. For

$$A = (10^{99}, 10^{-99}, -10^{99}), \quad B = (1, 1, 1), \tag{30}$$

the value of SPROD(A, B) is 0, of course, while SCALP(A, B, 0) gives the correct answer 10^{-99}. Persons who believe that multiple precision will solve all accuracy problems should determine how much precision is required on their machine for SPROD(A, B) to duplicate this result. Once satisfied, they can then try (31) below. Although these examples are extreme, vectors of the types one can expect to encounter in actual problems can be given for which SPROD is highly inaccurate.

Speed. It turns out that the accurate scalar product function SCALP is *faster* than SPROD if DIM > 1. The typical times given in Ref. [12] in milliseconds for a 2.5 MHz Z80 processor are:

SCALP 8 + 5.5(DIM),

SPROD 1 + 9.6(DIM).

Since all the matrix and matrix by vector multiplication routines are based on SCALP, they can be expected to execute faster than their inaccurate simulations in REAL arithmetic. Furthermore, the current implementation of Pascal-SC handles access to elements of arrays very efficiently by means of an "array descriptor" [18], from which addresses of elements can be calculated quickly.

Deferred overflow. Under ordinary circumstances, SCALP will not indicate an overflow unless the actual result is outside the range of representable floating-point numbers. For example, for

$$A = (10^{99}, 10^{-99}, -10^{99}), \quad B = (10^{99}, 1, 10^{99}), \tag{31}$$

SCALP(A, B, 0) will compute the correct result 10^{99}, while SPROD(A, B) will overflow for I = 1.

Ease of use. The function SCALP is just as easy to use as SPROD, and requires no additional source code in the program, since it is part of the Pascal-SC system. Furthermore, SCALP is more versatile, since its arguments can be arbitrary one-dimensional arrays of floating-point numbers of the same length, of which RVECTOR is only a special case. When called, SCALP consults the array descriptors of its arguments [18] to determine if they are in fact of equal length, and then calculates their scalar product if this is true. Thus, SCALP can be used to calculate scalar products of vectors of various dimensions in the same program.

COMPLEX AND INTERVAL FLOATING-POINT VECTOR AND MATRIX ARITHMETIC

The basic ideas here are the same as for real vector and matrix arithmetic: convenience, based on the use of operator notation for expressions, and accuracy. The library subroutines provided expect the type declarations

```
TYPE CVECTOR = ARRAY [DIMTYPE] OF COMPLEX;
     CMATRIX = ARRAY [DIMTYPE] OF CVECTOR;
```

and

```
TYPE IVECTOR = ARRAY [DIMTYPE] OF INTERVAL;
     IMATRIX = ARRAY [DIMTYPE] OF IVECTOR;
```

in the corresponding cases. For accurate scalar products, the respective functions

```
FUNCTION CSCALP (VAR A, B: CVECTOR; AKDIM: INTEGER): COMPLEX;
  EXTERNAL 188;
```

and

```
FUNCTION ISCALP (VAR A, B: IVECTOR; AKDIM: INTEGER): INTERVAL;
  EXTERNAL 88;
```

are provided. The functions form the basis of the subroutines for accurate matrix and matrix by vector multiplication. The products are computed from the first AKDIM components of each vector argument. This permits the flexibility of using vectors of various dimensions AKDIM < DIM in the same program. CSCALP computes the BPA for the scalar product of complex vectors; directed rounding is not provided for the complex scalar product or complex matrix and matrix by vector multiplications. ISCALP computes the smallest interval which contains the exact result, as for other interval operations [4].

SOLUTION OF LINEAR SYSTEMS OF EQUATIONS AND MATRIX INVERSION

These are problems which arise time after time in scientific, engineering, and statistical computation. The Pascal-SC system subroutines for these purposes, which use the accurate scalar product and interval arithmetic, yield results which are far more exact than can be obtained by ordinary floating-point arithmetic. Furthermore, guaranteed error bounds are given for results, so the reliability of the computation is immediately determinable. There are subroutines for both real and interval vectors and matrices.

The basic procedure in the system library for the solution of linear systems of equations is LGLP (an acronym for the German words for "linear equations solution program"). This procedure is declared by

```
PROCEDURE LGLP(DIM, AKDIM: INTEGER; VAR A: RMATRIX, VAR B: RVECTOR;
          VAR Y: IVECTOR);
  EXTERNAL 524;
```

[4]. The purpose of this procedure is to solve the linear system

$$Ax = B \tag{32}$$

with coefficient matrix A and right-hand side B. Instead of a floating-point approximation to the solution x, LGLP calculates an *interval vector* Y which, if proper contains the *exact* solution x of (32), and *proves* that the floating-point matrix A is nonsingular. This allows one to determine not only an approximate value for x, but also guaranteed error bounds for it [19]. The parameter AKDIM in the formal parameter list allows one to solve systems of size smaller than DIM if desired; only the first AKDIM rows and columns of A and components of B are involved in the calculation.

Failure of LGLP to return a proper interval vector Y indicates that A is singular or extremely ill-conditioned. In this case, the components of Y will be set equal to the improper interval $[+1, -1]$. A test should be made for this condition immediately on return from LGLP, since all interval subroutines expect proper intervals as data [4].

Thus, LGLP either gives a solution with guaranteed accuracy or an error indication. In practice, LGLP has been observed to succeed for well-known examples of badly conditioned matrices, such as Hilbert matrices [15, 20]. It *cannot* be fooled by singular coefficient matrices.

When one is solving several systems with the same matrix but different right sides, considerable time is saved if the approximate LU-decomposition of A and other preliminary calculations are only done once. The procedure LGLPR is available for this purpose.

Similar procedures LGLI and LGLIR are available for the case that the components of the coefficient matrix A and right side B are intervals, i.e. A is of type IMATRIX and B is of type IVECTOR. The interval vector Y in this case bounds all solutions of real systems with coefficient matrices belong to A and right sides belonging to B. This is helpful in case the data are subject to uncertainty.

For matrix inversion, the subroutine

```
PROCEDURE INVP (DIM,AKDIM: INTEGER; VAR A: RMATRIX; VAR C: IMATRIX);
  EXTERNAL 526;
```

will, if successful, compute an interval matrix C which contains the inverse of the real (point) matrix A. Singularity or extreme ill-condition of A is reported in the same way as for LGLP, while successful calculation of C proves that A is nonsingular, as before. Finally,

```
PROCEDURE INVI (DIM,AKDIM: INTEGER; VAR A, C: IMATRIX);
  EXTERNAL 527;
```

is used for inversion of interval-valued matrices. If C is computed successfully as an IMATRIX of proper intervals, then C contains the inverses of all real matrices contained in the interval matrix A. Return of C with all components equal to the improper interval $[+1, -1]$ indicates that A contains at least one singular or very badly conditioned real matrix.

On a microcomputer with 64 kB of storage, LGLP and LGLPR are limited to about DIM = 25 or less, LGLI, LGLIR, and INVP to DIM = 20, and INVI to DIM = 15. For these relatively small systems, the time required for execution seems to be reasonable.

EIGENVALUES AND EIGENVECTORS

The Pascal-SC system provides the standard subroutine

```
PROCEDURE EIGEN (DIM,AKDIM: INTEGER; VAR A: RMATRIX; LAMBDA: REAL;
                 VAR X: RVECTOR; VAR ILAMBDA: INTERVAL; VAR Y: IVECTOR);
  EXTERNAL 534;
```

for the calculation of guaranteed interval bounds for real eigenvalues and vectors of real matrices A, in particular, symmetric matrices. This is another type of calculation which occurs often in engineering and other scientific computation. In addition to the actual dimension AKDIM and the matrix A, EIGEN expects floating-point approximations LAMBDA and X to the eigenvalue and eigenvector of interest, or at least values with which to start the calculation. If successful, the interval value ILAMBDA and interval vector Y returned include an exact real eigenvalue and eigenvector of the floating-point matrix A, and furthermore guarantee that the included eigenvalue is of multiplicity one. Hence, EIGEN will not succeed for multiple eigenvalues [4, 20]. In case of failure, EIGEN will return improper intervals for ILAMBDA and the components of Y.

THE ACCURATE SUM OF n FLOATING-POINT NUMBERS

Statistical calculations, in particular, often require the computation of the sum of n floating-point numbers and usually also their squares,

$$S = \sum_{i=1}^{n} a_i, \quad T = \sum_{i=1}^{n} a_i^2, \tag{33}$$

In Pascal-SC, it is possible to compute the BPA for S and T, or round the result upward or downward to the closet floating-point number by taking the a_i as components of an RVECTOR A and using SCALP. For $E = (1, 1, 1, \ldots, 1)$, one has

$$\text{S = SCALP(A, E, ROUND): } \textit{and} \quad \text{T = SCALP(A, A, ROUND);} \tag{34}$$

with the desired best-possible result. However, in the case of the sum S, the standard Pascal-SC subroutine

```
FUNCTION SUM (VAR A: RVECTOR; ADKIM, ROUND: INTEGER): REAL;
  EXTERNAL 480;
```

is provided. This function performs the addition of the first AKDIM elements of A to the BPA or result of directed rounding of the BPA. SUM, like SCALP, uses the long accumulator, and the values of ROUND have the same significance as given earlier for SCALP. It is thus possible to call SUM again without clearing the long accumulator, to allow independent accumulation of partial sums without loss of accuracy. However, no other arithmetic operations are allowed between successive calls to SUM [4].

When computing sums of interval numbers, one can use

$$\text{IS := ISCALP(IA, IE, DIM);} \tag{35}$$

where IE has components all equal to [1, 1]. For the interval sum of squares, however, it is preferable to form the interval vector IQA with components equal to ISQR(A[I]) for I = 1..DIM, and then compute

$$\text{IT := ISCALP(IQA, IE, DIM);} \tag{36}$$

rather than ISCALP(IA, IA, DIM), for the reason given earlier about the preferability of ISQR(X) to X*X for intervals.

PROGRAMMING IN Pascal-SC

The only new techniques in Pascal-SC for a Pascal programmer to acquire are the definition and use of operators. Except for these, there is no difference between Pascal and Pascal-SC programming. Therefore, the discussion here will focus on the operator concept [3]. The definition of an operator subroutine is headed by

OPERATOR ⟨name⟩(⟨formal parameter list⟩)⟨result name⟩: ⟨type⟩;

The code following this heading is the same as for a function having the same purpose. The formal parameter list consists of one or two identifiers and their types. Thus, operators are either unary or binary. Since operators occur in expression strings ("infix" notation), their arguments have to be of expression type. That is, VAR A, etc., is not allowed in the formal parameter list. The examples of operators given can be used as models.

There are two ways to name an operator in Pascal-SC:

(i) By redefining ("overloading") one of the standard Pascal-SC operator symbols for a new data type or types.

(ii) By use of an arbitrary name selected by the user which conforms to the ordinary rules for identifiers in Pascal [1]. In this case (see below), the priority of the operator also has to be declared.

These two methods will be discussed separately.

Overloading standard operator symbols

This is the most common method used in scientific and engineering computing to name Pascal-SC operators, since one usually wishes to follow the ordinary mathematical notation encountered in the formulas being used. The standard operator symbols in Pascal-SC are, in order of decreasing priority:

Unary operators:

NOT, +(unary), −(unary)

Multiplicative (binary) operators:

*, /, DIV, MOD, AND, **, *>, *<, />, /<

Additive (binary) operators:

+, −, +>, +<, −>, −<, ++, OR

Relational (binary) operators:

=, <>, <=, >=, <, >, IN, ><

The fundamental distinction between a unary and a binary OPERATOR is that the formal parameter list for the operator contains exactly one parameter in the first case, and exactly two in the second, and these are the only possibilities. An overloaded operator will have the *same* priority as its symbol in the table above. In the case of + and −, the parameter list of the operator heading will specify whether they are unary (highest priority) or binary.

One convenience of Pascal-SC that is immediately apparent is that one can define ** to perform exponentiation on whatever numerical types are appropriate for the application at hand. However, this should be done with care. Some good methods are given in Ref. [21]. A simple recursive implementation of ** is:

```
OPERATOR**(R: REAL; K: INTEGER) RES: REAL;
  BEGIN
    IF K <= 0 THEN R := 1/R;
    IF K < 0 THEN K := -K;
    IF K = 0 THEN RES := 1 ELSE RES := A*(A**(K - 1));
  END;                                                    (37)
```

to perform R^K for integral powers of floating-point numbers. This operator makes it possible to write x^3, x^4, etc. as x**3, x**4, etc. in expressions to be evaluated, which is a more convenient way to represent these simple powers than by a procedure or function call. This operator is for the purpose of illustration; it is not necessarily the most efficient or accurate way to do exponentiation [21]. For intervals, the following operator will give better results for even powers:

```
OPERATOR**(I: INTERVAL; K: INTEGER) RES: INTERVAL;
  BEGIN
    IF K <= 0 THEN I := 1/I;
    IF K < 0 THEN K := -K;
    IF K = 0 THEN RES := INTPT(1)
      ELSE IF K = 1 THEN RES := I
        ELSE RES := ISQR(I)*(I**(K - 2))
  END;                                                    (38)
```

The standard Pascal-SC interval function INTPT(x) transforms the real number x into the degenerate interval $[x, x]$. [A function or procedure should be used to compute the result of raising an interval base to an interval power, since the operator ** is used to compute the intersection of INTERVAL variables, see (25).]

The order of the operands in the formal parameter list determines the order in which the operator will be applied. The compiler distinguishes various uses of the same operator symbol by the type(s) of its operand(s), and their order if the operator is binary. Thus, in the same program, "+" can be used to denote addition of complex numbers, intervals, vectors, matrices, quaternions, polynomials, etc., in addition to its standard meaning for integers and floating-point numbers. All that is required is that the appropriate definition of OPERATOR + be given in the heading of the program for each meaning of " + " in the body of the program.

Of course, the compiler recognizes only the rules of arithmetic for user-defined data types which are provided to it by the programmer. For example, if one wishes to use expressions in which variables of both type INTEGER and type GRADIENT [22] appear, both

```
OPERATOR + (K: INTEGER; G: GRADIENT)RES: GRADIENT;
```

and

```
OPERATOR + (G: GRADIENT; K: INTEGER)RES: GRADIENT;
```

must be defined in the heading of the program so that the compiler can produce code for both $K + G$ and $G + K$. Type GRADIENT consists of the value of a function together with its gradient vector, and is declared by

```
TYPE GRADIENT = RECORD F: REAL; DF: RVECTOR END;
```

in Pascal-SC [22]. In this case, both operators produce the same result, consisting of the alteration of the function value G.F of the GRADIENT variable G to K + G.F = G.F + K, respectively, with no change in the gradient vector G.DF [22]. However, it could happen that the user is working with quantities for which addition is not necessarily commutative. Pascal-SC allows the possibility of defining the result of " + " or any other binary operator to be dependent on the order of the operands.

Named operators

In Pascal-SC, the user can name operators according to the ordinary Pascal rules for identifies [1]. For example, a binary Boolean operator XOR for "exclusive or" could be defined by

```
OPERATOR XOR (A, B: BOOLEAN) RES: BOOLEAN;
  BEGIN
    RES := (A AND NOT B) OR (NOT A AND B);
  END;
```

A typical program statement using XOR would be

```
IF OBS1 XOR OBS2 THEN PROB := 0.25 ELSE PROB := 0.75;          (39)
```

which would assign the value 0.25 to PROB if just one of OBS1, OBS2 is TRUE, or 0.75 otherwise.

In order for the Pascal-SC compiler to recognize XOR as the name of an operator, and assign a priority to it, a PRIORITY declaration for it must follow directly after the heading line of the program, the line which gives the name of the program and the list of files used. From highest to lowest priority, these priority declarations have the forms

```
PRIORITY ⟨Operator name⟩ = @;   {Unary operators}
PRIORITY ⟨Operator name⟩ = *;   {Multiplicative operators}
PRIORITY ⟨Operator name⟩ = +;   {Additive operators}
PRIORITY ⟨Operator name⟩ = =;   {Relational operators}
```

Thus, suppose one writes a program called CHANCE which uses the operator XOR to calculate probabilities of outcomes in some stochastic model, and only the standard files INPUT and OUTPUT are used for communication between the program and the outside world. If XOR is to have the same priority as OR, then the first two lines in the heading of the source code for the program would be

```
PROGRAM CHANCE (INPUT, OUTPUT);
  PRIORITY XOR = +;
```

The standard sequence of definitions and declarations would then follow to complete the heading of the program, and then the body of the program consisting of the actual statements to be executed. The structure of a Pascal-SC program therefore differs only slightly from that of an ordinary Pascal program, as shown in the next section.

Structure of a Pascal-SC program

In Pascal-SC, the order of declarations in the heading is somewhat freer than in standard Pascal [3]. However, the general principle applies that everything must be declared or defined before use.

Since overloading standard operator symbols is more common than using named operators, the headings of most Pascal-SC programs will look identical to Pascal programs except for OPERATOR definitions. Programming is further simplified because Pascal-SC already provides operators for most of numerical data types commonly encountered in scientific and engineering computation, such as complex numbers, intervals, vectors, and matrices, as explained in the previous sections.

The difference between Pascal and Pascal-SC programs is most striking in the statements of the actual body of the program. Here, the power of operator notation makes it possible to write expressions clearly and compactly. This elimination of complicated sequences of function and procedure calls shortens programs and makes the source code much easier to read and understand. This facilitates documentation as well as use of the program.

With just two exceptions, noted below in boldface type, the sequence of a Pascal-SC program is identical to an ordinary Pascal program [1]:

```
PROGRAM ⟨Name⟩ (⟨List of internal file names⟩);
  PRIORITY
  LABEL
  CONST
  TYPE
  VAR                    ⟨Declarations and definitions⟩
  PROCEDURE
  FUNCTION
  OPERATOR
BEGIN
  ...  ⟨Statements comprising the body of the program⟩
END.
```

The order in which procedures, functions, and operators are declared is arbitrary, as in Pascal. The implementation of the operator concept in Pascal-SC is based on the fact that the underlying virtual machine (the so-called KL/P machine) can stack operands of arbitrary data types, so that functions with results of arbitrary type can be computed efficiently [18]. This refinement is permits considerable savings in the number of machine instructions actually needed, and hence leads to shorter execution times [18].

CONCLUSIONS

The accuracy of Pascal-SC arithmetic and the convenience of operator notation for manipulation of numerical and other data types make this language a valuable tool for scientific, engineering, and statistical computation. In addition to its usefulness for routine problems, such as solution of linear systems of equations, experience has shown that it is possible use this language to program and carry out some rather sophisticated computations, even on a microcomputer. Examples include numerical solution of a nonlinear integral equation [23], the solution of nonlinear systems of equations by iterative methods [22, 24], and the solution of ordinary differential equations by real and interval Taylor series [25]. In these applications the operator concept of Pascal-SC was used to implement automatic evaluation of derivatives and Taylor series for functions defined by expressions in ordinary mathematical notation [22, 25, 26]. The microcomputer systems used in these investigations can best be described as minimal: eight-bit machines with Z80 processors, 64 kB of main storage, and two disk drives. The Pascal-SC compiler used was developed by Professors Kulisch and Wippermann and their associates at the Universities of Karlsruhe and Kaiserslautern in Germany, and is described in Refs [2–4]. Even these modest resources appear adequate for many of the day-to-day calculations needed by engineers, scientists, and statisticians, as well as for research on methods in numerical analysis which can be applied to larger problems. As "personal computers" grow in size and speed, the accuracy and convenience of Pascal-SC will provide the user with a more powerful tool, and its features will also be advantageous on forthcoming larger machines.

Acknowledgements—The author would like to thank Professor George F. Corliss for reading the manuscript carefully, and a large number of valuable suggestions. Example (20) was provided by Professor Dr Ulrich Kulisch. Sponsored by the U.S. Army under Contract No. DAAG 29-80-C-0041.

REFERENCES

1. K. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd edn. Springer, Berlin (1978).
2. U. Allendörfer and R. Kirchner, *Pascal-SC Bedienungsanleitung* [Pascal-SC User Guide]. Department of Computer Science, University of Kaiserslautern (1982).
3. M. Neaga, *Pascal-SC Sprachbeschreibung und Programmieranleitung* [Pascal-SC Language Description and Programming Guide]. Department of Computer Science, University of Kaiserslautern (1982).
4. J. Wolff von Gudenberg, *Gesamte Arithmetik des Pascal-SC-Rechners. Benutzerhandbuch* [Complete Arithmetic of the Pascal-SC Computer User Handbook]. Institute for Applied Mathematics, University of Karlsruhe (1981).
5. G. Bohlender, K. Grüner, E. Kaucher, R. Klatte, W. Krämer, U. W. Kulisch, S. M. Rump, Ch. Ullrich, J. Wolff van Gudenberg and W. L. Miranker, Pascal-SC: a Pascal for contemporary scientific computation. Research Report RC 9009, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.
6. U. Kulisch and W. L. Miranker, *Computer Arithmetic in Theory and Practice*. Academic Press, New York (1981).
7. L. B. Rall, Accurate computer arithmetic for scientific computation. *Proceedings of the 1982 Army Numerical Analysis Conference*, pp. 343–356. U.S. Army Research Office, Research Triangle Park, N.C. (1982).
8. J. M. Yohe, Roundings in floating-point arithmetic. *IEEE Trans. Comput.* **C-22,** 577–586 (1973).
9. R. E. Moore, *Interval Analysis*. Prentice–Hall, Englewood Cliffs, N.J. (1966).
10. R. E. Moore, Methods and applications of interval analysis. *SIAM Studies in Applied Mathematics, 2.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia (1979).
11. H. Böhm, Evaluation of arithmetic expressions with maximum accuracy. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 121–137. Academic Press, New York (1983).
12. G. Bohlender and K. Grüner, Realization of an optimal computer arithmetic. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 247–268. Academic Press, New York (1983).
13. G. Alefeld and J. Herzberger, *Introduction to Interval Computations*. (Translated by J. Rokne). Academic Press, New York (1983).
14. R. E. Moore, New results on nonlinear systems. In *Interval Mathematics 1980* (Edited by K. L. E. Nickel), pp. 165–180. Academic Press, New York (1980).
15. S. M. Rump, Computer demonstration packages for standard problems of numerical mathematics. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 27–49. Academic Press, New York (1983).
16. U. Kulisch, A new arithmetic for scientific computation. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 1–26. Academic Press, New York (1983).
17. U. Kulisch and G. Bohlender, Features of a hardware implementation of an optimal arithmetic. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 269–290. Academic Press, New York (1983).
18. R. Kirchner, Überblick über die vorliegende Implementierung der Pascal-Spracherweiterung [Overview of the Current Implementation of the Pascal Language Extension]. Department of Computer Science, University of Kaiserslautern (1981).
19. L. B. Rall, Representation of intervals and optimal error bounds. *Math. Comp.* **41,** 219–227 (1983).
20. S. M. Rump, Solving algebraic problems with high accuracy. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 51–120. Academic Press, New York (1983).
21. W. F. Cody Jr and W. Waite, *Software Manual for the Elementary Functions*. Prentice–Hall, Englewood Cliffs, N.J. (1980).
22. L. B. Rall, Differentiation in Pascal-SC: type GRADIENT. *ACM Trans. Math. Software* **10,** 161–184 (1984).
23. L. B. Rall, Interval methods for fixed-point problems. MRC Technical Summary Report No. 2583. University of Wisconsin–Madison (1983). To appear in *Contemp. Math.*
24. A. Cuyt and L. B. Rall, Computational implementation of the multivariate Halley method. MRC Technical Summary Report NO. 2481, University of Wisconsin–Madison (1983). To appear in *ACM Trans. Math. Software.*
25. G. Corliss and L. B. Rall, Automatic generation of Taylor series in Pascal-SC: basic operations and applications to ordinary differential equations. MRC Technical Summary Report No. 2497, University of Wisconsin–Madison (1983).
26. L. B. Rall, Differentiation and generation of Taylor coefficients in Pascal-SC. In *A New Approach to Scientific Computation* (Edited by U. Kulisch and W. L. Miranker), pp. 291–309. Academic Press, New York (1983).

APPENDIX

This is the actual Pascal-SC program used to calculate the results given in (23).

```
PROGRAM WEVAL(INPUT, OUTPUT);
$USES INTERVAL; {Refers the compiler to the INTERVAL library}
   VAR w, x, y: REAL; W, X, Y: INTERVAL; C: CHAR;
OPERATOR**(R:REAL; K:INTEGER) RES:REAL;
   BEGIN
     IF K< =0 THEN R:= 1/R;
     IF K<0 THEN K:= -K;
     IF K=0 THEN RES:= 1
           ELSE RES:= R*(R**(K - 1))
```

```
    END;
OPERATOR**(I:INTERVAL; K:INTEGER) RES:INTERVAL;
    BEGIN
      IF K < =0 THEN I:= 1/I;
      IF K<0 THEN K:= -K;
      IF K=0 THEN RES:= INPUT(1)
            ELSE IF K=1 THEN RES:= I
                  ELSE RES:= ISQR(I)*(I**(K-2))
  END;

BEGIN {Program WEVAL}
    C:= 'Y';WHILE C='Y' DO
    BEGIN {REAL and INTERVAL evaluation}
        WRITELN('Enter x, y:');READ(x, y); {Enter values}
        X:= INTPT(x);Y:= INTPT(y); {Convert REAL values to intervals}
        w:= 9*(x**4)-y**4+2*(y**2); {Calculate (21a)}
        W:= 9*(X**4)-Y**4+2*(Y**2); {Calculate (22a)}
        WRITELN('(a) w=',w); {Output results of (a)}
        WRITELN('    W=[',W.INF,',',W.SUP,']');
        WRITELN;
        w:= (3*(x**2)-y**2)*(3*(x**2)+y**2)+2*(y**2); {Calculate (21b)}
        W:= (3*(X**2)-Y**2)*(3*(X**2)+Y**2)+2*(Y**2); {Calculate (22b)}
        WRITELN('(b) w=',w); {Output results of (b)}
        WRITELN('    W=[',W.INF,',',W.SUP,']');
        WRITELN;
        WRITELN('More values (Y/N)?');READ(C, C)
    END {Evaluation}
END. {Program WEVAL}
```