Numerik mit Fortran 90

Sind genaue und zuverlässige Ergebnisse erzielbar?

Wolfgang V.Walter
Institut für Angewandte Mathematik
Universität Karlsruhe

Der neue Fortran–Standard bringt für das Programmieren im allgemeinen und für die Numerik im besonderen eine Vielzahl von Neuerungen. So sind vor allem die neuen Möglichkeiten der Feldverarbeitung, dynamische Felder, Zeiger, Module und benutzerdefinierte Datentypen und Operatoren für das zeitgemäße Programmieren unerläßlich.

Die vordefinierten Feldoperatoren und das neue Typ- und Operatorkonzept ermöglichen auch für Vektoren, Matrizen und andere Datenstrukturen eine natürliche, mathematische Schreibweise von Ausdrücken und Zuweisungen. Für die Numerik sind daneben die neuen vordefinierten Funktionen von Bedeutung. Hierbei handelt es sich nicht etwa um weitere mathematische Funktionen, sondern hauptsächlich um numerische Abfragefunktionen, numerische (und Bit-) Manipulationsfunktionen und Feldfunktionen. So kann man nun unter anderem die Characteristika der Zahldarstellung in den verschiedenen Zahlformaten erfragen, Mantisse und Exponent einer Gleitkommazahl extrahieren oder zusammensetzen und die Nachbarn einer Gleitkommazahl bestimmen. Außerdem sind alle Standardtypen mit einem Parameter zur Auswahl einer der (maschinenabhängigen) Darstellungsarten ausgestattet, so daß zum Beispiel die verwendete Zahldarstellung (Rechengenauigkeit) sehr einfach geändert werden kann.

Die erwähnten Hilfsmittel können bei richtiger Verwendung die Portabilität von Fortran 90 Programmen wesentlich verbessern. Leider ist dies aber keine Garantie für portable oder wenigstens einigermaßen kompatible Rechenergebnisse — erst recht nicht für mathematisch sinnvolle und zuverlässige numerische Lösungen. Über die mathematischen Eigenschaften der arithmetischen Operatoren, der Standardfunktionen und anderer vordefinierter Funktionen wird auch im neuen Fortran 90 Standard (wie in FORTRAN 77) nichts ausgesagt. Insbesondere fehlen Genauigkeitsanforderungen jeglicher Art.

So sind berechnete Ergebnisse nach wie vor von Hardware, Compiler, Laufzeitsystem, Code-Optimierung und anderen Faktoren abhängig. Die neuen vordefinierten Funktionen SUM, DOTPRODUCT und MATMUL sind aus numerischer Sicht besonders gefährlich. Hier kann durch Auslöschung innerhalb weniger Operationen das Ergebnis derart verfälscht werden, daß es mit der tatsächlichen Lösung weder Vorzeichen noch Größenordnung gemein hat. Berechnungen dieser Art sollten immer auf Auslöschungseffekte hin überprüft werden, es sei denn die Implementierung liefert garantierte Fehlerschranken.

Werkzeuge zur genauen und automatisch verifizierten Lösung numerischer Probleme sind zwar in Fortran 90 nicht vorhanden, können aber dank der vordefinierten numerischen Abfrage- und Manipulationsfunktionen in reinem Fortran 90 portabel geschrieben werden. Dies war in FORTRAN 77 nicht möglich. Es wird eine Bibliothek von Fortran 90 Modulen vorgestellt, mit deren Hilfe sich die erwähnten Probleme oft beheben oder umgehen lassen. Genaue und zuverlässige numerische Ergebnisse sind aber in Fortran 90 nur mit einem hohen Zusatzaufwand zu erreichen.

FORTRAN–XSC

A Portable Fortran 90 Module Library for Accurate and Reliable Scientific Computing

Wolfgang V. Walter
Institut für Angewandte Mathematik
Universität Karlsruhe

The new Fortran standard [14], developed under the name *Fortran 8x*, now known as *Fortran 90*, was finally adopted and published as an international standard in the summer of 1991. Fortran 90 offers a multitude of enhancements and extensions of the old FORTRAN 77 language [2]. Among the most prominent new features are parameterized intrinsic data types, extensive array handling facilities, dynamic arrays, user-defined data types and operators, generic interfaces, pointers, and modules. The new concepts greatly improve Fortran's potential for well-structured and portable programming.

Numerically, however, the Fortran 90 standard is still deficient since the mathematical properties of the arithmetic operators and mathematical functions, in particular any accuracy requirements, remain unspecified. Thus, computational results still cannot be expected to be compatible when using different computer systems with different floating-point units, compilers, and compiler options. Even the use of floating-point processors which all satisfy the same standard (e.g. IEEE 754 [3]) does not help much. As a consequence, portability and reliability of numerical results is still extremely difficult to achieve. In this respect there has been little improvement over the situation in FORTRAN 77.

A Fortran 90 module library called FORTRAN–XSC ("Fortran extension for scientific computing") is presented that attempts to remedy these problems. FORTRAN–XSC is a versatile numerical toolbox intended for use in a wide range of engineering/scientific applications. The main objective of FORTRAN–XSC is to provide a highly portable foundation for analyzing and improving the accuracy and reliability of numerical application programs. In particular, FORTRAN–XSC is designed to facilitate the development of numerical algorithms which deliver automatically verified results of high accuracy. For such algorithms there is no need to perform an error analysis by hand.

The FORTRAN–XSC library is written entirely in standard-conforming Fortran 90 [14] and is fully portable. It automatically adapts to the selected native floating-point system of the computer on which it is compiled and exploits the hardware arithmetic whenever possible. The library consists of a number of Fortran 90 modules featuring accurate scalar, vector and matrix arithmetic for real and complex numbers and intervals, accurate conversion routines for numeric constants and input/output data, multiple precision arithmetic, reliable and highly accurate versions of the Fortran 90 intrinsic functions SUM, DOTPRODUCT, MATMUL, and more. The user is given full control of the rounding mode to be used in an operation. The result of every operation is optimal with respect to the selected rounding, so all operations are guaranteed to be accurate to 1 ulp.

to be published in Computing Supplementum, 1993

1 Introduction

The common programming languages attempt to satisfy the needs of many diverse fields. While trying to cater to a large user community, these languages fail to provide specialized tools for specific areas of application. Thus the user is often left with ill-suited means to accomplish a task. In recent years, this has become quite apparent in numerical programming and scientific computing. Even though programming has become more convenient through the use of more modern language concepts, numerical programs have not necessarily become more reliable. This is true even if "good" floating-point arithmetic (e. g. which conforms to the IEEE Standard 754 for Binary Floating-Point Arithmetic [3]) is employed.

At the Institute of Applied Mathematics at the University of Karlsruhe there has been a long-term commitment to the development of programming languages suited for the particular needs of numerical programming. With languages and tools such as PASCAL-XSC [19], C-XSC, ACRITH-XSC [12, 34], and ACRITH [10, 11], the emphasis is on accuracy and reliability in general and on automatic result verification in particular.

In the 1980's, the programming language FORTRAN-SC [5, 24] was designed as a FORTRAN 77 extension featuring specialized tools for reliable scientific computing. It was defined and implemented at the University of Karlsruhe in a joint project with IBM Germany. The equivalent IBM program product *High Accuracy Arithmetic — Extended Scientific Computation*, called ACRITH-XSC for short, was released for world-wide distribution in 1990 [12]. Numerically it is based on IBM's *High-Accuracy Arithmetic Subroutine Library* (ACRITH) [10, 11], a FORTRAN 77 library which was first released in 1984.

When Fortran 90 finally became a standard in 1991, it was tempting to define another language extension for scientific computing. However, in order to avoid having to write another compiler, it was decided that the Fortran 90 language was convenient and powerful enough to allow the implementation of most of the desired features within the language — something completely impossible in FORTRAN 77.

Therefore, FORTRAN-XSC is made up of a set of portable Fortran 90 modules — a versatile toolbox for accurate and reliable numerical computation. In particular, FORTRAN-XSC is designed to facilitate the development of numerical algorithms with *automatic result verification*. Such algorithms deliver results of high accuracy which are verified to be correct by the computer. For example, self-validating numerical techniques have been successfully applied to a variety of engineering problems in soil mechanics, optics of liquid crystals, ground-water modelling and vibrational mechanics where conventional floating-point methods have failed.

The elementary arithmetic operations for real and complex numbers are available with a choice of five different rounding modes: *nearest* (to the nearest floating-point number), *upwards* (towards $+\infty$), *downwards* (towards $-\infty$), *towards zero* (truncation), and *away from zero* (augmentation). Additionally, the FORTRAN-XSC package offers the corresponding interval operations which are essential for auto-

matic result verification. Besides the elementary arithmetic operations, routines for the conversion of numerical constants and input/output data are provided.

The new Fortran 90 intrinsic functions SUM, DOTPRODUCT, and MATMUL are usually unreliable because they may suffer from cancellation of leading digits. Therefore, an accurate and fully reliable alternative implementation using a long fixed-point accumulator is offered by FORTRAN–XSC. Accurate accumulation is essential in many algorithms to attain high accuracy. The accurate dot product provides the foundation on which all vector/matrix products are built. Furthermore, it may serve as a basis for a highly accurate implementation of the Basic Linear Algebra Subroutines (BLAS), an "industry standard" defining a set of commonly used vector/matrix operations [26]. For all of the aforementioned operations and functions, the results are always optimal with respect to the selected rounding. This implies that their error is never more than 1 ulp (1 unit in the last place).

The Fortran 90 code is designed to allow a certain degree of optimization by the compiler, especially vectorization. On certain architectures, code optimization can be further improved by using special compiler directives or language extensions.

FORTRAN–XSC is closely related to the following standards:

Fortran 90 International Standard: Information technology – Programming languages – Fortran, ISO/IEC 1539 : 1991 (E) [14]

IEEE 754 IEEE Standard 754 for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754–1985 [3]

IEEE 854 IEEE Standard 854 for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Std 854–1987 [4]

LIA–1 Draft International Standard: Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic, ISO/IEC CD 10967-1 : 1992 (formerly called Language Compatible Arithmetic Standard (LCAS)) [16] *(in preparation)*

The design of FORTRAN–XSC has also been influenced by other research in the area of computer arithmetic. The simulation of double-precision or higher precision arithmetic in software has been the subject of a number of investigations. Some of the early work was done by Møller [25], Kahan [17], Dekker [8] and Linnainmaa [23]. A recent summary which appeals to hardware manufacturers to make exact floating-point operations accessible to the users is given in [6]. A vectorizable FORTRAN 77 version of the arithmetic runtime operations needed to run ACRITH–XSC [12] was designed and implemented by Schmidt [28]. A rigorous mathematical definition of computer arithmetic and roundings is given by Kulisch and Miranker in [21, 22].

2 Principles and Requirements

The module library FORTRAN-XSC consists of a number of Fortran 90 modules providing various arithmetic operations and other fundamental tools for a given floating-point system (which is usually provided by the hardware). At the Fortran 90 level, this floating-point system is identified by its kind type parameter value `fpkind`. The corresponding intrinsic types are `REAL(fpkind)` and `COMPLEX(fpkind)`.

Since FORTRAN-XSC automatically adapts to the specified underlying floating-point system, an extension to other floating-point systems is straightforward. All one has to do is to specify a different kind type parameter value `fpkind` and recompile all the modules. However, in order to avoid name clashes, global names such as module names and derived (data) type names must be changed before recompiling. If several floating-point systems are to be supported at the same time, conversion routines (with rounding control) must also be provided in order to be able to convert from one to the other.

2.1 Floating-Point System

A floating-point system $F = F(b, p, minexp, maxexp)$ is defined by the following characteristics: its base (or radix) b, its precision (or mantissa length) p, and its exponent range, bounded by the minimal exponent $minexp$ and the maximal exponent $maxexp$. These characteristics can be obtained in a portable fashion in Fortran 90 via the intrinsic inquiry functions `RADIX`, `DIGITS`, `MINEXPONENT`, and `MAXEXPONENT`, respectively. The letter F will be employed indiscriminately for a floating-point system and for the set of floating-point numbers it defines.

A floating-point number x in F is either 0, or it consists of a sign, a fixed-length mantissa with digits $d_1, d_2, \ldots, d_p$, and an exponent e with $minexp \le e \le maxexp$:

$$x = \pm\, b^e \sum_{i=1}^{p} d_i\, b^{-i}$$

According to this definition, the (radix) point is just to the left of the first digit of the mantissa. This convention is not important, however, since this can always be achieved by formally shifting the exponent range. The mantissa is given in base b notation, i. e. all $d_i \in \{0, \ldots, b-1\}$. A floating-point number is **normalized** if its leading mantissa digit d_1 is nonzero. Normalized floating-point numbers have a unique representation. The Fortran 90 standard uses "model numbers", which are normalized (or zero) by definition, to speak of floating-point numbers. A **denormalized** floating-point number is characterized by $d_1 = 0$ and $e = minexp$. Denormalized numbers are uniformly distributed around 0 (between the largest negative and the smallest positive normalized floating-point number).

The exponent of a floating-point number x (in the sense above) will be denoted by $e(x)$. It can be obtained via the Fortran 90 intrinsic function `EXPONENT(x)`. Also, the notation

$$\mathrm{ulp}(x) := b^{e(x)-p}$$

is used to mean 1 unit in the last place of a floating-point number x. In Fortran 90, the intrinsic function `SPACING(x)` can be employed to determine 1 ulp relative to x. For any floating-point number x, $x+\mathrm{ulp}(x)$ and $x-\mathrm{ulp}(x)$ are the neighboring floating-point numbers above and below x, unless one of them overflows or underflows, or unless x is an integral power of the base of the floating-point system. In the latter case, if $x = b^n$ for some integer n, then there are $b-1$ floating-point numbers between $x-\mathrm{ulp}(x)$ and x. By symmetry to zero, an analogous statement holds for $x = -b^n$. The immediate neighbor above/below any floating-point number x can be obtained in a portable way by calling the Fortran 90 intrinsic function `NEAREST(x,d)` with a positive/negative d, respectively.

Many common floating-point systems allow denormalized numbers, for example both IEEE Standards [3, 4]. This additional characteristic of a floating-point system cannot be determined in any portable way in Fortran 90 since the Fortran 90 standard uses "model numbers" — which are normalized by definition — when speaking of floating-point numbers. By default FORTRAN–XSCassumes floating-point numbers to be normalized, but the use of denormalized numbers is allowed if the underlying floating-point system supports them. The flag *denorm* is provided to indicate whether denormalized numbers are allowed or not (LIA–1 [16] makes a similar provision). However, the user has to explicitly set the flag *denorm* to true to tell FORTRAN–XSC that denormalized numbers are supported.

2.2 Rounding

A rounding is a special mapping from the real numbers $\mathbf{R}$ onto the floating-point numbers F. Any rounding $\bigcirc$ must satisfy the following two conditions:

1. The floating-point numbers F must be invariant under the rounding $\bigcirc$:

$$\bigcirc(s) = s \quad \text{for all } s \in F.$$

2. The rounding $\bigcirc$ must be a monotonic function on the real numbers $\mathbf{R}$:

$$x \leq y \Rightarrow \bigcirc(x) \leq \bigcirc(y) \quad \text{for all } x, y \in \mathbf{R}.$$

In other words, a rounding is a *monotonic, nondecreasing projection* from $\mathbf{R}$ onto F. This ensures that a rounding is always accurate to 1 ulp (one unit in the last place of the mantissa of a floating-point number). In particular, if a real number x is already representable in F, any rounding must return that same value x. If

a real number x is not representable in F, a rounding must return one of the two neighboring floating-point numbers (either the one above or the one below x).

Today, most computers provide at least so-called "faithful" arithmetic, which at least guarantees 1 ulp accuracy. However, traditionally, the monotonicity rule is not generally observed by the hardware arithmetic of computers.

Whenever an operation requires a rounding, the following five *roundings* (or *rounding modes*) are available in FORTRAN-XSC:

N	Nearest (to the nearest floating-point number)
Z	towards Zero (truncation)
I	towards Infinity (away from zero, augmentation)
U	Upwards Δ (upwardly directed, to $+\infty$)
D	Downwards ∇ (downwardly directed, to $-\infty$)

Except for the rounding *away from zero*, these are the same roundings that are required by the IEEE Standards. Also, the first three rounding modes are sign-symmetric: $\bigcirc(-x) = -\bigcirc(x)$, whereas the last two are not. Rather, the last two satisfy the symmetry rule: $\nabla(-x) = -\Delta(x)$ (or, equivalently, $\Delta(-x) = -\nabla(x)$).

2.3 System Requirements

In order for FORTRAN-XSC to function properly, the system on which it is compiled and run must satisfy some minimal requirements. The most important of these is that the intrinsic floating-point arithmetic provided by the system (typically in hardware) be *faithful*. This means that the four elementary floating-point operations $+, -, *, /$ must be accurate to 1 ulp. Note that 1 ulp (least bit) accuracy implies that whenever the mathematically exact result of an operation is representable as a floating-point number, the computed result is *exact*. Thus the maximum error is always *less than* 1 ulp. This guarantees that certain operations will be performed without error (e. g. addition of two floating-point numbers with the same exponent and opposite signs).

Least bit accuracy of the elementary machine operations is crucial in many places in the module library. On the other hand, a particular rounding mode is not required. In fact, it may well be that the machine "rounding" is not monotonic. In practice, the above accuracy requirement excludes only very few of the modern machine architectures (e. g. Cray vectorprocessors, which do not provide faithful arithmetic).

Another prerequisite concerns the Fortran 90 compiler (and compiler options) with which the FORTRAN-XSC module package is translated. The Fortran 90 compiler must respect parentheses in expressions, that is, the intended order of evaluation must be preserved. There are a number of places in the module library where any change to the order of evaluation would have numerically disastrous effects. Unfortunately, this requirement will sometimes preclude the use of high optimization

levels. Note, however, that a Fortran 90 processor that violates the integrity of parentheses does not conform to the Fortran 90 standard. Despite all this, large parts of the library are vectorizable.

For FORTRAN-XSC to function properly, it is also important that the Fortran 90 intrinsic functions for numeric inquiry and floating-point manipulation plus some others work without fail. FORTRAN-XSC has to rely on these functions (see section 6).

3 Arithmetic Modules

3.1 Exact Floating-Point Operations

In FORTRAN-XSC, the design principle for elementary arithmetic is to implement the exact operations first since they are the most general, and to base the operators with rounding control on the exact operations. For the four arithmetic operations $+, -, *, /$ it is possible to define and implement **exact floating-point operations** via subroutines with four floating-point arguments each: two input and two output arguments, all in the same floating-point format. In this context, **exact** means that the computed result is mathematically correct, that is, no information about the true mathematical result of the operation is lost. As long as no exception (such as overflow, underflow, or division by zero) occurs, the result of each of these floating-point operations can be represented without error by a pair of floating-point numbers in the same format as the operands.

The set of subroutines for exact floating-point arithmetic is summarized in the following table:

subroutine name	arguments in	arguments out	mathematical specification
`add_exact`	`(x, y, h, l)`		$x + y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
`sub_exact`	`(x, y, h, l)`		$x - y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
`mul_exact`	`(x, y, h, l)`		$x * y = h + l$ with $e(l) \leq e(h) - p$ unless $l = 0$
`div_exact`	`(x, y, q, r)`		$x = qy + r$ with $\|r\| < \|y\| \cdot \mathrm{ulp}(q)$ unless $q = 0$

In the case of addition, subtraction and multiplication, the high-order part h and the low-order part l are floating-point numbers whose mathematical sum is the exact result of the operation and which do not overlap, i. e. their exponents differ by at least p (the precision or mantissa length), unless $l = 0$. In the case of division, a partial quotient q and the corresponding exact remainder r are produced, again both floating-point numbers. The exponent difference $e(x) - e(r)$ must sometimes

be allowed to be $p-1$, while at other times it is required to be at least p. Essentially, this means that the relationship must be $|r| < |y| \cdot \mathrm{ulp}(q)$ and not $|r| < \mathrm{ulp}(x)$.

A proof of the fact that the result of these operations can always be represented by two floating-point numbers (unless an exception occurs) can be found in [6].

3.2 Elementary Arithmetic

Fortran 90, as opposed to ACRITH-XSC [12] and FORTRAN-XSC, does not provide any means for automatic error control or for deliberate rounding control. In particular, the arithmetic operators with directed roundings `+<`, `+>`, `-<`, `->`, `*<`, `*>`, `/<`, `/>`, which are predefined in ACRITH-XSC, are not available in Fortran 90. Thus, regrettably, Fortran 90 does not provide access to the rounded floating-point operations defined by the IEEE Standards 754 and 854 [3, 4]. In view of the steadily increasing number of processors conforming to these standards, this is most unfortunate for the whole numerical community.

The elementary arithmetic operations $+, -, *, /$ for real (and complex) numbers are available with the five rounding modes provided by FORTRAN-XSC. Alternatively, the rounding mode can be set by calling a special subroutine named `set_rounding`. The elementary operations are then accessible via the generic operators `.ADD.`, `.SUB.`, `.MUL.`, and `.DIV.`, applying the rounding specified in the last call to `set_rounding`. As long as the rounding has never been set by the user, it defaults to *nearest*.

Experience shows that a default rounding mode is generally useful only to compute *approximations*. The upwardly and downwardly directed roundings, on the other hand, are typically used in pairs to do interval arithmetic. Unfortunately, most IEEE floating-point processors do not provide the arithmetic operations with the rounding mode integrated into the instruction code. Rather, they require a special instruction to change the rounding mode to be used in subsequent operations. Note that switching the rounding mode to *upwards* before evaluating an expression does not generally deliver an upper bound to the value of the expression by any means, nor will *downwards* deliver a lower bound.

The following table lists the elementary operators with rounding control:

generic	Nearest	Zero	Infinity	Up	Down
`.ADD.`	`.ADDN.`	`.ADDZ.`	`.ADDI.`	`.ADDU.`	`.ADDD.`
`.SUB.`	`.SUBN.`	`.SUBZ.`	`.SUBI.`	`.SUBU.`	`.SUBD.`
`.MUL.`	`.MULN.`	`.MULZ.`	`.MULI.`	`.MULU.`	`.MULD.`
`.DIV.`	`.DIVN.`	`.DIVZ.`	`.DIVI.`	`.DIVU.`	`.DIVD.`

Of course, on any particular machine featuring IEEE arithmetic in hardware, the elementary arithmetic operations with one of the rounding modes *nearest*, *towards zero*, *upwards*, and *downwards* may be implemented using the hardware operations for greater efficiency. This cannot be done exclusively in standard-conforming

Fortran 90, however. Thus FORTRAN-XSC does not currently provide any special support for IEEE hardware. However, an adaptation is certainly possible.

FORTRAN-XSC also provides rounding control for the conversion of numeric constants and input/output data. This ensures that the user knows exactly what data enters the computational process and what data is produced as a result. Besides the default rounding, the monotonic downwardly and upwardly directed roundings, symbolized by < and >, respectively, are available. For further details, refer to section 4.

3.3 Complex Arithmetic

Complex addition and subtraction are trivially reduced to the corresponding real operations. Complex multiplication requires the accurate evaluation of an expression of the form $ab \pm cd$. Such expressions are sometimes called *short dot products*. Because of the danger of cancellation of leading digits, exact double-length products need to be computed and added/subtracted with sufficient accuracy to allow correct rounding. For reasons of efficiency and because it is a relatively frequent operation occurring in many applications, the short dot product is included as a special operation. Whenever a dot product involves only two terms, this operation can be employed for greater efficiency.

Complex division is rather intricate and requires careful implementation. A special algorithm is needed to obtain sufficient accuracy for correct rounding.

3.4 Interval Arithmetic

By controlling the rounding error at each step of a calculation, it is possible to compute guaranteed bounds on a solution and thus verify numerical results on the computer. Enclosures of a whole set or family of solutions can be computed using interval arithmetic, for example to treat problems involving imprecise data or other data with tolerances, or to study the influence of certain parameters. Interval analysis is particularly valuable for stability and sensitivity analysis. It provides one of the essential foundations for reliable numerical computation.

FORTRAN-XSC provides complete interval arithmetic consisting of the derived types `INTERVAL` and `COMPLEX_INTERVAL`, arithmetic and relational operators, and the necessary type conversion functions. The result of every arithmetic operation is accurate to 1 ulp.

An interval is represented by a pair of (real or complex) numbers, its infimum (lower bound) and its supremum (upper bound). For the infimum, the direction of rounding is always downwards, for the supremum, upwards, so that the inclusion property is never violated. By adhering to this principle, the computed result interval will and must always contain the true solution set.

The arithmetic interval operators +, -, *, / as well as the operators .IS. (intersection) and .CH. (convex hull) are provided. The relational operators for intervals are the standard comparison operators and the operators .SB. (subset), .SP. (superset), .DJ. (disjoint), .IN. (point contained in interval), and .INT. (point or interval contained in interior of interval).

In order to be able to access interval bounds, to compose intervals and to perform various other data type changes, type conversion functions such as INF (infimum), SUP (supremum), and IVAL are available. Other useful functions include MID (midpoint of an interval), RADIUS, and DIAM (diameter (width) of an interval).

The arithmetic operations for real and complex intervals are implemented using the elementary arithmetic operations with upwardly and downwardly directed rounding to compute the infimum and the supremum. However, complex interval multiplication and division require more sophisticated algorithms. FORTRAN-XSC also provides a special notation for real and complex intervals and routines for the conversion of interval constants and input/output data. For details, refer to section 4.

3.5 Vector/Matrix Arithmetic

In traditional programming languages such as FORTRAN 77, Pascal, or Modula-2, each vector/matrix operation requires an explicit loop construct or a call to an appropriate subroutine. Unnecessary loops, long sequences of subroutine calls, and explicit management of loop variables, index bounds and intermediate result variables complicate programming enormously and render programs virtually incomprehensible.

Fortunately, the situation has improved a lot with Fortran 90. Fortran 90 offers extensive array handling facilities such as allocatable arrays, array pointers, subarrays (array sections), various intrinsic array functions, and predefined array operators. All array operators are defined as element-by-element operations in Fortran 90. This definition has the advantage of being uniform, but the disadvantage that highly common operations such as the dot product (inner product) of two vectors or the matrix product are not easily accessible. The Fortran 90 standard does not provide an operator notation for these operations, and it prohibits the redefinition of an intrinsic operator (e.g. *) for an intrinsically defined usage. Instead, the dot product is only accessible through the intrinsic function call DOTPRODUCT(V,V), the other vector/matrix products through the intrinsic function calls MATMUL(V,M), MATMUL(M,V), and MATMUL(M,M). Clearly, function references are far less readable and less intuitive than operator symbols, especially in complicated expressions.

If one wants to reference the intrinsic functions DOTPRODUCT and MATMUL via an operator notation, there are only two choices: either one defines a new operator symbol, say .MUL., for all possible type combinations that can occur in vector/matrix multiplication, or one defines new data types, e.g. RVECTOR, DRVECTOR, CVECTOR, DCVECTOR, RMATRIX, ... and then overloads the operator symbol * for all possible type combinations of these new types. Both of these methods are quite cumbersome

and seem to contradict one of the major goals of the Fortran 90 standard, namely to cater to the needs of the numerical programmer, in particular by providing extensive and easy-to-use array facilities. Note that both of these methods require a minimum of 64 operator definitions to cover all of the intrinsic cases. If more than two `REAL` and two `COMPLEX` types (single and double precision) are provided by an implementation, this number becomes even larger.

The following Fortran 90 intrinsic functions are numerically critical because of the accumulation of intermediate rounding errors and the possibility of severe cancellation in the summation process:

function	operation performed
`SUM`	summation of the components of a vector
`DOTPRODUCT`	dot product of two vectors
`MATMUL`	product of two matrices or of a vector and a matrix

The most serious drawback of these Fortran 90 intrinsics is that they are generally unreliable numerically. Since the Fortran 90 standard lacks any kind of accuracy requirements, it seems inevitable that different implementors will implement these functions differently. Even worse is the fact that *any* traditional floating-point summation technique is sensitive to the order of summation and is sure to fail in ill-conditioned cases because leading-digit cancellation may completely destroy the result. On vector processors, the problem is compounded by automatic vectorization by the compiler. The user has virtually no influence on the order in which the accumulation is performed. Typically, on pipelined processors, several partial sums are first computed and then added to form the final result. In the process, the summands are completely scrambled.

Now that these critical functions have been "formally standardized" (but not numerically), the potential danger to the user becomes very evident. For the Fortran 90 programmer, these functions appear to be very welcome since they seem to provide a portable way of specifying these highly common operations, especially as they are inherently difficult to implement. However, the user has no knowledge or control of the order in which the accumulation is performed. This makes any kind of realistic error analysis virtually impossible.

The inevitable consequence of this situation is that these three new intrinsic functions are unusable for all practical purposes — at least if one wishes to write portable Fortran 90 programs which deliver reliable results. Tests on large vector computers show that simple rearrangement of the components of a vector or a matrix can result in vastly different results [9, 27]. Different compilers with different optimization and vectorization strategies and different computational modes (e. g. scalar mode or vector mode with a varying number of vector pipes) are often responsible for incompatible and unreliable results.

As an example, consider the computation of the trace of the $n \times n$ product matrix

C of a $n \times k$ matrix A and a $k \times n$ matrix B, which is defined by

$$\text{trace}\,(C) = \text{trace}\,(A \cdot B) = \sum_{i=1}^{n} C_{ii} = \sum_{i=1}^{n} \sum_{j=1}^{k} A_{ij} * B_{ji} \quad .$$

In ACRITH-XSC [12] this double sum can be calculated by the following so-called *dot precision expression*:

```
TRACE = #*( SUM(A(i,:)*B(:,i), i = 1, n) )
```

The notation is simple and efficient and the computed result is guaranteed to be accurate to 1/2 ulp in every case.

In contrast, the corresponding Fortran 90 program looks something like this:

```
TRACE = 0.0
DO I = 1, N
   TRACE = TRACE + DOTPRODUCT(A(I,:), B(:,I))
END DO
```

Here the computational process involves on the order of $2nk$ rounding operations if, as is typical in the computation of dot products, the products are rounded before they are added and the accumulation is performed in the same floating-point format in which the elements of `A` and `B` are given. Far more critical is the fact that cancellation can, and often will, occur during summation. This leads to results of unknown accuracy at best, or to completely wrong and meaningless results if many leading digits cancel. Since the Fortran 90 standard does not impose any accuracy requirements on intrinsic functions such as `SUM`, `DOTPRODUCT`, and `MATMUL`, there are no simple remedies.

Thus, unless an implementation gives explicit error bounds for these intrinsic functions, every Fortran 90 programmer should think twice before using them, especially if the possibility of leading digit cancellation cannot be excluded.

For the above reasons, FORTRAN-XSC provides an alternative, highly accurate implementation of these functions which are so fundamental to most branches of mathematics. In order to be able to compute the dot product of arbitrary vectors with 1 ulp accuracy, the summation of the exact double-length products is performed without error in a *long fixed-point accumulator*. Such an accumulator covers twice the exponent range of the underlying floating-point system in order to accomodate all possible double-length products. In the case of FORTRAN-XSC, it is implemented as a sequence of floating-point segments (instead of integer segments as is the case in ACRITH [10, 11], ACRITH-XSC [12], and PASCAL-XSC [19]) because it is assumed that this will improve the performance of the elementary dot product operations. However, this assumption may be false for certain modern RISC processors. The exact (unrounded) result of an accumulation process can

be stored to full accuracy in a variable of type `DOT_PRECISION` or rounded to a floating-point number using one of the five available roundings.

The *dot product* for real vectors and for real interval vectors are both elementary in the sense that the interval case cannot be reduced to the real case in any simple way. The complex and complex interval dot products, on the other hand, are easily reduced to the corresponding real case. Analogously, two routines are necessary for the *summation* of real numbers and real intervals. Again, the complex cases are reducible to their real analogues.

All vector/matrix products are implemented via the accurate dot product and produce results which are accurate to 1 ulp in every component. Additionally, FORTRAN-XSC provides the arithmetic element-by-element operations with rounding control for real and complex vectors and matrices. The operators (and rounding modes) are the same as for real and complex numbers. The arithmetic element-by-element operations for vectors and matrices with interval and complex interval components are also available.

4 Data Conversion for Input/Output

FORTRAN-XSC provides routines for accurate conversion of numerical data from one base to another and from a character string to one of the elementary data types of FORTRAN-XSC (for input) and vice versa (for output). All of the non-interval conversions are available with a choice of five roundings. For interval data, the rounding is always to the smallest possible interval enclosing the given interval.

On input, the constant given in the input string may be specified with an arbitrary number of digits in any base in the range 2–36. The letters `A-Z` are used to represent the digits 10–35, respectively. A provision for bases greater than 36 has not been made. It is assumed that any other base of interest is a power of one of those provided. The conversion uses as many digits as are necessary to determine the correctly rounded internal floating-point number or interval. On output, an arbitrary number of digits may be requested by the user. The length, the base and the rounding of the output constant can be chosen by the user.

The following functions are available for input:

```
REAL (string, rounding)

CMPLX (string, rounding)

IVAL (string)

CIVAL (string)
```

The functions for output are of the form

```
STR (number, base, length, rounding)
```

where number may be of type `REAL`, `COMPLEX`, `INTERVAL`, or `COMPLEX_INTERVAL`.

For the conversion from string to string, the following function is provided:

```
STR (string, base, length, rounding)
```

In some sense, this function is the ultimate conversion routine. It is capable of correctly converting a constant specified with an arbitrary number of digits in a given base to any other base, generating the prescribed number of digits in the target number system while respecting the rounding mode.

In the above functions, the type of the argument `string` and the result type of all functions with generic name `STR` is `VARYING_STRING`. This derived type for varying-length character strings is defined in module `ISO_VARYING_STRING`, whose functional definition is to become a collateral standard (currently draft international standard ISO/IEC CD 1539-1 [15]) to the Fortran 90 standard [14]. This module provides elementary tools for working with fully dynamic character strings of arbitrary length.

Non-advancing (stream) I/O is (finally!) available in Fortran 90. This enables reading and writing of partial records, making it possible to define one's own I/O in a portable and flexible way. In combination with the above routines, input/output with rounding control is straightforward.

Reading from left to right, the syntax of a constant is as follows. The constant may be optionally preceded by a `+` or `-` sign. The mantissa is specified as a sequence of digits which may or may not include a (radix) point. The base (radix) is given in decimal notation and is appended to the mantissa with a `%` sign as separator. The default for the base is 10 (decimal). If the base is greater than 10 and the first significant digit of the mantissa is represented by a letter, i. e. at least 10, then the mantissa should be preceded by an extra zero. The exponent is introduced by the letter `E` or `D` (for compatibility with Fortran), which may be uppercase or lowercase. The value of the exponent is given by an optionally signed integer. The sign, the (radix) point, the base, and the exponent are optional. The base and the exponent are always in decimal notation.

The parameter `rounding` is optional. If it is omitted on output, the rounding to be applied is the default rounding. If it is omitted on input, the rounding may be specified within the constant notation in the argument `string`. If a rounding is specified in the input string, the rounding symbol (`<` for *downwards* or `>` for *upwards*) must precede the constant. The whole notation should be parenthesized in this case. If no rounding is specified at all, the rounding used is the default rounding currently in effect.

Examples of the accepted syntax are:

constant	interpretation
`1001101%2`	binary constant = 77 (decimal)
`-76.50%8`	octal constant = 62.625 (decimal)
`+0FFA000%16E-4`	hexadecimal constant = 255.625 (decimal)
`(<-3.14159265E+000)`	decimal constant rounded downwards
`(>0Z.YX4%36E2)`	base 36 constant rounded upwards = 46617.111...

For intervals, no rounding can be specified. The rounding is always to the smallest enclosing interval. There is a special notation for intervals:

constant	interpretation
`(<-2.00001,-1.99999>)`	real interval enclosing -2
`(<-3.1415926535898>)`	optimal (1 ulp wide) enclosure of $-\pi$ (approx.)
`((<2.9,3.1>),(<1E0,1>))`	complex interval with Re around 3 and $Im = 1$
`(<(2.9,1E0),(3.1,1)>)`	same complex interval as above

5 Exception Handling

It is an unfortunate fact that Fortran 90 does not provide any exception handling facilities. Thus it is impossible to treat exceptions which occur during runtime in any simple and portable way — if at all. This is particularly detrimental to numerical applications, where exceptional cases are quite common. FORTRAN-XSC has to live with this situation and tries to preclude the possibility of numerical exceptions such as overflow, underflow, and division by zero in the floating-point operations it uses by preliminary testing and scaling as appropriate.

Among others, the following arithmetic exceptions are recognized by FORTRAN-XSC:

exception	meaning
`invalid_op`	invalid operation
`div_by_zero`	division by zero
`overflow`	exponent overflow
`underflow`	exponent underflow
`inexact`	result of an "exact" operation is inexact

The IEEE Standards 754 and 854 [3, 4] define the same five types of exceptions, whereas LIA-1 [16] does not distinguish *division by zero* and *invalid operation* and does not recognize the exception *inexact*.

Trap handling can be enabled or disabled for each exception individually. All exceptions are enabled by default. During runtime, the user may inquire about the occurrence of disabled exceptions at any point in time. It is also possible to define the manner in which an exception is to be treated by a user-written trap handling routine.

6 Useful Fortran 90 Intrinsic Functions

Besides the numeric inquiry functions `RADIX`, `DIGITS`, `MINEXPONENT`, and `MAXEXPONENT` which are needed to determine the main characteristics of a floating-point system, the following inquiry functions are sometimes useful:

function	result
`EPSILON`	Wilkinson epsilon (largest rel. dist. betw. 2 adjacent fl-pt numbers)
`HUGE`	largest floating-point number (maxreal)
`TINY`	smallest positive normalized floating-point number (minreal)

For the low-level treatment of floating-point numbers, the following floating-point manipulation functions are particularly helpful in ensuring better portability of the Fortran 90 code:

function reference	result
`EXPONENT(x)`	exponent of floating-point number x (base b)
`FRACTION(x)`	fraction (mantissa) of floating-point number x
`SET_EXPONENT(x,e)`	floating-point number x with exponent replaced by e
`SCALE(x,k)`	floating-point number x multiplied by b^k
`NEAREST(x,d)`	floating-point neighbor of x in direction of sign of d
`SPACING(x)`	1 ulp relative to floating-point number x

None of these intrinsic functions were available in FORTRAN 77. This made a portable implementation of floating-point software impossible in FORTRAN 77. Large parts of FORTRAN-XSC depend on these intrinsics. If they are not absolutely reliable, the module library cannot function properly.

7 Conclusion and Outlook

In the foreword of the Fortran 90 standard [14], under the heading "Numerical computation", one finds the following statement: "Scientific computation is one of the principal application domains of Fortran, and a guiding objective for all of the technical work is to strengthen Fortran as a vehicle for implementing scientific software." In a modern programming language whose declared goal is to support scientific applications, a clear specification of the exact mathematical behavior and accuracy of the arithmetic operators and intrinsic functions should be an integral part of the language definition. Contrary to its claim, the Fortran 90 standard does not really provide much better support in this domain than did FORTRAN 77. Fortran 90 does, however, provide the necessary language tools which enable a software library such as FORTRAN-XSC to provide such numerical support. Unfortunately, as long as there is no adequate language and hardware support, a large performance penalty seems inevitable.

In the presence of a growing number of floating-point processors conforming to standards such as the IEEE Standard 754 for Binary Floating-Point Arithmetic [3] or the IEEE Standard 854 for Radix-Independent Floating-Point Arithmetic [4], the reluctance of programming language standardization committees to provide easy access to the elementary arithmetic operations with directed roundings by incorporating special operator symbols such as `+<`, `+>`, `-<`, `->`, `*<`, `*>`, `/<`, `/>` is incomprehensible. Since none of the major programming languages provide any simple means to access these fundamental operations, it is not astonishing that they are seldom used. Interval arithmetic is one way of making these operations accessible and more widely accepted.

The IMACS-GAMM Resolution on Computer Arithmetic [13] requires that all arithmetic operations — in particular the compound operations of vector computers such as "multiply and add", "accumulate", and "multiply and accumulate" — be implemented in such a way that guaranteed bounds are delivered for the deviation of the computed floating-point result from the exact result. A result that differs from the mathematically exact result by at most 1 ulp (i.e. by just one rounding) is highly desirable and always obtainable, as demonstrated by various existing implementations [10, 11, 12, 34, 19]. The "Proposal for Accurate Floating-Point Vector Arithmetic" [7] essentially requires the same mathematical properties for vector operations as are required for the elementary arithmetic operations by the IEEE Standards. Hopefully, such user requests will influence the hardware design of computing machinery, especially of supercomputers [18, 20], in the near future. Accuracy requirements are also highly desirable in programming language standards to make scientific software more portable in the numerical sense.

In its current state, FORTRAN-XSC provides only the essential foundations for accurate and reliable numerical computing. With the fundamental modules tested and running, things are only beginning to become interesting. An alternative, highly accurate implementation of the Basic Linear Algebra Subroutines (BLAS) [26] is planned for the near future. Modules providing arithmetic for varying-precision numbers and intervals, for polynomials, and for truncated Taylor series are soon to follow. At the same time, fundamental application programs such as solvers for linear and nonlinear systems and for ordinary differential equations are being written to test the module library and to broaden the basis of FORTRAN-XSC.

Acknowledgement

My special thanks go to Enrik Berkhan and Arno Schauman who did (and are still doing) most of the programming. The implementation of FORTRAN-XSC would not have been possible without their relentless efforts. I would also like to thank my colleagues at the Institute of Applied Mathematics for their willingness to discuss any problems and answer any questions. I am particularly indebted to my colleague Dr.-Ing. Lutz Schmidt for his invaluable advice on vectorization and the software simulation of computer arithmetic. Above all, I would like to thank Prof. Kulisch for creating a fruitful research atmosphere at the Institute of Applied

Mathematics and for giving me the opportunity to pursue this project and to attend many national and international Fortran 90 standardization meetings.

References

[1] Adams, E.; Kulisch, U. (eds.): *Scientific Computing with Automatic Result Verification.* Academic Press, Orlando, 1992.

[2] American National Standards Institute: *American National Standard Programming Language FORTRAN.* ANSI X3.9–1978, 1978.

[3] American National Standards Institute / Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754–1985, New York, 1985.

[4] American National Standards Institute / Institute of Electrical and Electronics Engineers: *IEEE Standard for Radix-Independent Floating-Point Arithmetic.* ANSI/IEEE Std 854–1987, New York, 1987.

[5] Bleher, J. H.; Rump, S. M.; Kulisch, U.; Metzger, M.; Ullrich, Ch.; Walter, W. (V.): *FORTRAN-SC: A Study of a FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH.* Computing **39**, 93–110, Springer, 1987.

[6] Bohlender, G.; Kornerup, P.; Matula, D. W.; Walter, W. V.: *Semantics for Exact Floating Point Operations.* Proc. of 10th IEEE Symp. on Computer Arithmetic (ARITH 10) in Grenoble, 26.6.–28.6.1991, 22–26, IEEE Computer Society, 1991.

[7] Bohlender, G.; Cordes, D.; Knöfel, A.; Kulisch, U.; Lohner, R.; Walter, W. V.: *Proposal for Accurate Floating-Point Vector Arithmetic.* In [1], 87–102, 1992.

[8] Dekker, T. J.: *A Floating-Point Technique for Extending the Available Precision.* Numerical Mathematics **18**, 224–242, 1971.

[9] Hammer, R.: *How Reliable is the Arithmetic of Vector Computers?* In [29], 467–482, 1990.

[10] IBM *System/370 RPQ, High-Accuracy Arithmetic.* SA22-7093-0, IBM Corp., 1984.

[11] IBM *High-Accuracy Arithmetic Subroutine Library (ACRITH)*, General Information Manual. 3rd ed., GC33-6163-02, IBM Corp., 1986.

. . . , Program Description and User's Guide. 3rd ed., SC33-6164-02, IBM Corp., 1986.

[12] IBM *High Accuracy Arithmetic — Extended Scientific Computation (ACRITH-XSC)*, General Information. GC33-6461-01, IBM Corp., 1990.

. . . , Reference. SC33-6462-00, IBM Corp., 1990.

. . . , Sample Programs. SC33-6463-00, IBM Corp., 1990.

. . . , How to Use. SC33-6464-00, IBM Corp., 1990.

. . . , Syntax Diagrams. SC33-6466-00, IBM Corp., 1990.

[13] IMACS, GAMM: *Resolution on Computer Arithmetic.* In Mathematics and Computers in Simulation **31**, 297–298, 1989; in Zeitschrift für Angewandte Mathematik und Mechanik **70**, no. 4, p. T5, 1990; in Ch. Ullrich (ed.): *Computer Arithmetic and Self-Validating Numerical Methods*, 301–302, Academic Press, San Diego, 1990; in [29], 523–524, 1990; in E. Kaucher, S. M. Markov, G. Mayer (eds.): *Computer Arithmetic, Scientific Computation and Mathematical Modelling*, IMACS Annals on Computing and Appl. Math. **12**, 477–478, J.C. Baltzer, Basel, 1991.

[14] International Standards Organization: *Information technology – Programming languages – Fortran.* Int. standard ISO/IEC 1539:1991(E), 1991.

[15] International Standards Organization: *Varying Length Character Strings in Fortran.* Draft int. std. ISO/IEC CD 1539-1 (collateral std. to ISO/IEC 1539:1991(E)), 1992.

[16] International Standards Organization: *Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic.* Draft int. std. ISO/IEC CD 10967-1:1992, 1992.

[17] Kahan, W.: *Further Remarks on Reducing Truncation Errors.* Commun. ACM **8**, 40, 1965.

[18] Kirchner, R.; Kulisch, U.: *Arithmetic for Vector Processors.* Proc. of 8th IEEE Symp. on Computer Arithmetic (ARITH 8) in Como, 256–269, IEEE Computer Society, 1987.

[19] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL-XSC Sprachbeschreibung mit Beispielen.* Springer, Berlin, Heidelberg, 1991.

...: *PASCAL-XSC Language Reference with Examples.* Springer, Berlin, Heidelberg, 1992.

[20] Knöfel, A.: *Fast Hardware Units for the Computation of Accurate Dot Products.* Proc. of 10th IEEE Symp. on Computer Arithmetic (ARITH 10) in Grenoble, 70–74, IEEE Computer Society, 1991.

[21] Kulisch, U.; Miranker, W. L.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

Kulisch, U.: *Grundlagen des numerischen Rechnens: Mathematische Begründung der Rechnerarithmetik.* Reihe Informatik **19**, Bibl. Inst., Mannheim, 1976.

[22] Kulisch, U.; Miranker, W. L. (eds.): *A New Approach to Scientific Computation.* Notes and Reports in Comp. Sci. and Appl. Math., Academic Press, Orlando, 1983.

[23] Linnainmaa, S.: *Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums.* BIT **14**, 167–202, 1974.

[24] Metzger, M.; Walter, W. (V.): *FORTRAN-SC: A Programming Language for Engineering/Scientific Computation.* In [29], 427–441, 1990.

[25] Møller, O.: *Quasi Double-Precision in Floating-Point Addition.* BIT **5**, 37–50, 1965.

[26] NAG: *Basic Linear Algebra Subprograms (BLAS).* The Numerical Algorithms Group Ltd, Oxford, 1990.

[27] Ratz, D.: *The Effects of the Arithmetic of Vector Computers on Basic Numerical Methods.* In [29], 499–514, 1990.

[28] Schmidt, L.: *Semimorphe Arithmetik zur Automatischen Ergebnisverifikation auf Vektorrechnern.* Ph. D. thesis, Univ. Karlsruhe, 1992.

[29] Ullrich, C. (ed.): *Contributions to Computer Arithmetic and Self-Validating Numerical Methods.* IMACS Annals on Computing and Appl. Math. **7**, J.C. Baltzer, Basel, 1990.

[30] Walter, W. (V.): *FORTRAN 66, 77, 88, -SC . . . — Ein Vergleich der numerischen Eigenschaften von FORTRAN 88 und FORTRAN-SC.* ZAMM **70**, 6, T584–T587, 1990.

[31] Walter, W. V.: *Flexible Precision Control and Dynamic Data Structures for Programming Mathematical and Numerical Algorithms.* Ph. D. thesis, Univ. Karlsruhe, 1990.

[32] Walter, W. V.: *Fortran 90: Was bringt der neue Fortran-Standard für das numerische Programmieren?* Jahrbuch Überblicke Mathematik 1991, 151–175, Vieweg, Braunschweig, 1991.

[33] Walter, W. V.: *A Comparison of the Numerical Facilities of FORTRAN-SC and Fortran 90.* Proc. of 13th IMACS World Congress on Computation and Appl. Math. (IMACS '91) in Dublin, Vol. **1**, 30–31, IMACS, 1991.

[34] Walter, W. V.: *ACRITH-XSC: A Fortran-like Language for Verified Scientific Computing.* In [1], 45–70, 1992.