Recent Developments in Fortran

1. FORTRAN–XSC, a Fortran 95 Library for Accurate and Reliable Scientific Computing
2. Fortran Standardization
3. Current Situation and Future of Fortran

Wolfgang V. Walter
Institut für Wissenschaftliches Rechnen
Technische Universität Dresden
D – 01062 Dresden

wwalter@math.tu-dresden.de

History of FORTRAN–XSC

- ACRITH: High-Accuracy Arithmetic Subroutine Library
  developed and implemented at Inst. of Applied Mathematics, Univ. of Karlsruhe in collaboration with IBM R & D, Böblingen released as IBM Program Product 5664-185 in 1984/85/86

- FORTRAN–SC: FORTRAN 77 Extension for Scientific Computation
  developed and implemented at Inst. of Applied Mathematics, Univ. of Karlsruhe for IBM R & D, Böblingen since 1984

- ACRITH–XSC: High Accuracy Arithmetic – Extended Scientific Computation (equivalent to FORTRAN–SC)
  released as IBM Program Product 5684-129 in 1990

- Fortran 90: International Standard ISO/IEC 1539:1991

- Fortran 95: International Standard ISO/IEC 1539:1997
  completed in 1996 and published in 1997

- Fortran 200x: First Committee Draft
  public review ended on December 27, 2002,
  final publication expected before end of 2004

- FORTRAN–XSC: A Portable Fortran 95 Module Library for Accurate and Reliable Scientific Computing
  under development at Institute of Scientific Computing, Technical University of Dresden

Overview of FORTRAN–XSC

- Fortran 95 eXtension for Scientific Computing
- fully portable Fortran 95 module library
- versatile toolbox for accurate and reliable numerical calculations
- automatic adaptation to native floating-point arithmetic
- works with or without IEEE hardware support
- operator interfaces for natural, math-like notation
- arithmetic of 1 ulp accuracy (1/2 ulp when rounding to nearest)
- tools for rounding error control and interval analysis
- tools for handling leading digit cancellation
- tools for automatic differentiation
- solvers for some fundamental numerical problems

Components of FORTRAN–XSC

- real and complex arithmetic with choice of 5 rounding modes
- real and complex interval arithmetic to compute enclosures
- accurate accumulation of sums and dot products
- determination of number of cancelled leading digits in an accumulation
- accurate vector/matrix arithmetic (real, complex, interval, and complex interval)
- exact floating-point operations (approximation plus error/remainder)
- varying-length multiple-precision arithmetic (with automatic memory management)
- polynomial arithmetic including accurate evaluation
- Taylor polynomial, gradient, and Hessian arithmetic
- accurate conversion of numerical constants with full rounding control
- general base conversion of numbers (arbitrary source and target base, length, and rounding mode)
- input/output for real, complex, interval, and complex interval with optimal rounding

Implementation and Portability

FORTRAN–XSC is a Fortran 95 module library. At the general user level, it features a modular design and a user-friendly interface through derived types, dynamic arrays, and overloaded functions and operators. To the extent to which this is possible in Fortran 95, an object-oriented approach was adopted.

However, FORTRAN–XSC also allows access to lower levels where efficiency is more important than a modular design, and simple interfaces, e.g. operators, are often impossible or unwanted. Most of the elementary arithmetic tools, especially the basic operations for multiple-precision arithmetic and accurate sums and dot products, are implemented in one large base module.

To port FORTRAN–XSC to a new platform, all one should have to do is:

1. copy the Fortran 95 source code to the new system,
2. select the underlying floating-point format to which FORTRAN–XSC is to adapt by specifying its kind type parameter value `fpkind`,
3. compile all FORTRAN–XSC modules in the proper order.

FORTRAN–XSC is written in such a way that the only necessary change to the source code is the specification of the kind type parameter value `fpkind`.

Of course, a fully standard-conforming Fortran 95 system is a necessity.

System Requirements

Faithful Native Floating-Point Arithmetic:

The floating-point hardware operations $+, -, *, /$ must deliver results of least bit (1 ulp) accuracy.

Integrity of Parentheses:

In arithmetic expressions, parentheses must be respected, i. e. the Fortran 95 compiler must not perform optimizations which are not standard-conforming.

Reliability of Fortran 95 Intrinsic Functions:

The new Fortran 95 intrinsic functions for floating-point manipulation (and some others) must be absolutely reliable:

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

32-Bit Integers:

FORTRAN–XSC currently assumes that integers with at least 32 bits are available and that these are the default kind. Furthermore, FORTRAN–XSC assumes that all exponent calculations can be done in integer.

Numerical Shortcomings of Fortran

- Fortran standard fails to specify the mathematical properties and the numerical behavior of arithmetic operations and mathematical functions
- in particular, the Fortran standard does not contain any minimal accuracy requirements for the arithmetic operators, the mathematical functions, or the conversion of numerical constants and input/output data
- Fortran still lacks tools for the accurate accumulation of sums and dot products
- particularly dangerous are the intrinsic functions `SUM`, `DOT_PRODUCT` and `MATMUL` where severe cancellation of leading digits may occur
- user has virtually no control of the rounding error in a computation
- Fortran 200x still does not provide direct access to the IEEE operations with directed roundings toward $-\infty$ and $+\infty$ (∇ and Δ) as defined by the IEEE Standard 754 for Binary Floating-Point Arithmetic, e. g. by providing the operators:
  +<, +>, -<, ->, *<, *>, /<, />
- thus it is still very difficult to compute reliable bounds on a solution, and even more difficult to compute tight interval enclosures
- computational results still depend on hardware, compiler, runtime system, code optimization, vectorization, and parallelization
- large effort required to make Fortran 90/95/200x numerically reliable
- (most of this is also true for other programming languages)

Arithmetic Operations

Exact Floating-Point Operations:

The result of these operations is always exact, i. e. no information is lost (unless an exception occurs). For this reason, they are used as elementary building blocks for other arithmetic operations.

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

Rounded Floating-Point Operations:

The rounding mode can be set globally or selected in each operation via the specific operator names. If a generic operator name is used, the rounding mode is the one currently in effect. The default rounding mode is to Nearest.

generic	Nearest $\bigcirc$	toward Zero $\sqcup$	Away from zero $\sqcap$	Up $\triangle$	Down $\triangledown$
.ADD.	.ADDN.	.ADDZ.	.ADDA.	.ADDU.	.ADDD.
.SUB.	.SUBN.	.SUBZ.	.SUBA.	.SUBU.	.SUBD.
.MUL.	.MULN.	.MULZ.	.MULA.	.MULU.	.MULD.
.DIV.	.DIVN.	.DIVZ.	.DIVA.	.DIVU.	.DIVD.
.DOT.	.DOTN.	.DOTZ.	.DOTA.	.DOTU.	.DOTD.

The operators in the last row perform dot products (inner products) of vectors and matrices.

Interval Operations:

The derived types INTERVAL and COMPLEX_INTERVAL are provided with complete interval arithmetic. Each interval operation produces an optimal and guaranteed enclosure (accurate to 1 ulp) of the solution set.

+	-	*	/	.o.
.ADD.	.SUB.	.MUL.	.DIV.	.DOT.

The symbolic operators +, −, ∗, / have the same meaning as the corresponding named operators. The interval dot product .DOT. requires special implementation. Unfortunately, Fortran 95 does not provide a symbolic operator for this common operation.

Various operators such as .ISECT. (intersection) and .CHULL. (convex hull), the comparison operators, and other relational operators such as .SB. (subset), .SP. (superset), .IN. (element of) and .INT. (in interior) are also available.

Interval Functions:

Among others, the following functions for the derived types INTERVAL and COMPLEX_INTERVAL are provided:

```
IVAL(x)
IVAL(x,y)
INF(ival)
SUP(ival)
MID(ival)
RADIUS(ival)
DIAM(ival)
ABS(ival)
```

Vector/Matrix Intrinsic Functions:

Due to the possibility of severe cancellation of leading digits, the following Fortran 95 intrinsics are numerically critical and generally unreliable:

```
DOT_PRODUCT(a,b)

MATMUL(A,b)

MATMUL(a,B)

MATMUL(A,B)

SUM(array, dim, mask)
```

where `a` and `b` are vectors and `A` and `B` are matrices, and where `array` is an arbitrary array whose elements are summed along dimension `dim` corresponding to the true elements of `mask` (`dim` and `mask` are optional).

For these functions, FORTRAN–XSC provides an alternative implementation which always delivers results of 1 ulp accuracy. A "long accumulator" is used to calculate arbitrary sums and dot products without error.

Vector/Matrix Operators:

All dot product operations can also be accessed via the operators `.DOT.`, `.DOTN.`, `.DOTZ.`, `.DOTA.`, `.DOTU.`, and `.DOTD.`, e. g.:

```
a .DOT. b

A .DOT. b

a .DOT. B

A .DOT. B
```

All other vector/matrix operators work on an element-by-element basis and are based on the corresponding scalar operations. The same operator notation as in the scalar case is used.

Elementary Dot Precision Operations:

A number of elementary operations are provided for more direct access to the "long accumulator". A long accumulator can be stored in a R_DOT variable (RI_DOT, C_DOT, and CI_DOT are also available).

```
TYPE(R_DOT)    ::  dot
REAL(fpkind)   ::  x, y
INTEGER        ::  rounding

DOT_INIT(dot)                  ! dot= 0
DOT_ADD(dot, x)                ! dot + x
DOT_ADD(dot, x, y)             ! dot + x*y
DOT_SUB(dot, x)                ! dot - x
DOT_SUB(dot, x, y)             ! dot - x*y
DOT_MUL(dot, x)                ! dot * x
DOT_ROUND(dot, rounding)       ! round(dot)
```

The final result delivered by DOT_ROUND is always accurate to 1 ulp (1/2 ulp when rounding to nearest).

The **multiplication** of a long accumulator by a floating-point number is a relatively rare, but useful operation. For example, it allows the exact evaluation of polynomials.

Also, with this set of dot precision operations, all BLAS (Basic Linear Algebra Subprograms) can be implemented with 1 ulp (1/2 ulp) accuracy.

Conversion of Numerical Constants

Input:

```
REAL (string, rounding, final_pos)
CMPLX (string, rounding, final_pos)
IVAL (string, final_pos)
CIVAL (string, final_pos)
```

Output:

```
STR (real, base, length, rounding)
STR (complex, base, length, rounding)
STR (interval, base, length)
STR (complex_interval, base, length)
```

String to String Conversion:

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

Constants may be specified in any base $b \in \{2, 3, \ldots, 36\}$ by appending the base to the mantissa in the form $\ldots\%b$. Conversion between any of these bases is possible. The default base is 10.

All arguments except the first are optional. For input and for string to string conversion, the rounding may either be specified in the constant notation in the string or by the argument `rounding`. If neither is specified, the global rounding mode currently in effect is used. If both are specified, they must coincide. For output, the rounding is either specified by the argument `rounding`, or it defaults to the global rounding mode.

The argument `final_pos` indicates the position of the first character after the constant in the string.

Examples of Rounded Constants:

In constants, only the rounding modes Downward (∇) (indicated by <) and Upward (Δ) (indicated by >) can be explicitly specified.

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 downward
`(>0Z.YX4%36E2)`	base 36 const. rd. upward $= 46617.111\ldots$

Examples of Interval Constants:

For interval constants, the rounding mode is always Downward (∇) for the infimum (lower bound) and Upward (Δ) for the supremum (upper bound).

constant	interpretation
`(<-2.00001,-1.99999>)`	real interval enclosing -2
`(<-0.0001%3>)`	optimal (1 ulp wide) enclosure of $-1/81$
`((<2.9,3.1>),(<1E0,1>))`	complex interval (*Re* around 3, $Im = 1$)
`(<(2.9,1E0),(3.1,1)>)`	same complex interval as above

Current Status, Applications, Problems

- FORTRAN–XSC also contains solvers for
  - ODE's (by Dietrich, based on Lohner's AWA)
  - Gauss and Romberg integration
  - linear optimization (infeasible interior-point method)
  - global optimization
  - other fundamental problems, e.g. linear systems
- portability compromised by unreliable Fortran 95 compilers
- maintenance and support difficult

Overview of Fortran 90 Features

- Fortran 90 is a superset of FORTRAN 77
- all FORTRAN 77 programs are valid Fortran 90 programs and should compile and run as expected
- complete modernization of Fortran (subject to strict compatibility with FORTRAN 77)
- major additions, extensions and improvements in all areas
- many new features and concepts:
  - free source form
  - improved, block-structured syntax
  - array processing
  - assumed-shape arrays
  - allocatable arrays
  - dynamic storage
  - pointers
  - optional arguments
  - recursion
  - modules
  - derived types (user-defined data structures)
  - user-defined operators and assignment
  - explicit and generic interfaces
  - procedure, operator and assignment overloading

Other Fortran 90 Features

- RECURSIVE functions and subroutines
- RESULT variable for function result
- INTENT(IN), INTENT(OUT), INTENT(INOUT) arguments
- OPTIONAL arguments
- PRESENT inquiry function to test for presence of optional arguments
- argument keywords (dummy argument names for argument identification)
- non-advancing I/O to read/write partial records (ADVANCE='NO')
- NAMELIST I/O to read/write data with variable names
- most FORTRAN 77 intrinsic functions are now elemental, i. e. applicable to arrays (no loops required)
- many new intrinsic functions for numeric manipulation and inquiry, array construction, reduction, reshaping, manipulation, inquiry, ...
- bit intrinsics for INTEGER (MIL-STD-1753)
- intrinsic subroutines for random numbers, bit copying, date and time
- binary, octal, and hexadecimal constants and I/O for INTEGER
- INCLUDE line to include source text (not a statement)

Language Evolution

The concept of language evolution was first introduced by the Fortran 90 standard. The intent was to modernize the language by slowly eliminating old, obsolescent features which could be replaced by better, more modern features or notations. However, in practice, users do not want to change their legacy codes, and thus compilers are forced to keep all features — even those that were officially deleted in the Fortran 95 language.

- deleted features:
  - in Fortran 90, no features were deleted from FORTRAN 77
  - in Fortran 95, the following features, which were previously declared obsolescent, were deleted:
    - real and double precision DO variables
    - branching to END IF from outside the block
    - PAUSE statement
    - ASSIGN, assigned GOTO and assigned FORMAT specifier
    - H edit descriptors and Hollerith constants
- remaining obsolescent features (there is no concrete plan to delete these):
  - arithmetic IF
  - shared DO loop termination (several loops use the same label)
  - DO termination on statements other than END DO or CONTINUE
  - alternate return

Overview of Fortran 95

Major New Features:

- FORALL statement and construct
- PURE and ELEMENTAL user-defined procedures
- Implicit initialization of derived-type objects
- Initial association status for pointers

Some Minor New Features:

- automatic deallocation of ALLOCATABLE arrays
- nested WHERE constructs and masked ELSEWHERE
- new intrinsic function NULL()
- new intrinsic function CPU_TIME
- small changes in a few intrinsic functions

Main Goals for Fortran 200x

Fortran 200x is designed to be a language

- for high performance numerical, scientific and engineering programming
- with high quality data abstraction and user extensibility features.

Two ISO/IEC Technical Reports have already been published; their contents will be integrated into Fortran 200x:

1. IEEE Floating Point Exception Handling (via procedures)
2. Enhanced Data Type Facilities (extends use of ALLOCATABLE arrays)

The target date for publication of the Fortran 200x standard is November 2004.

There are also 2 optional parts in the family of Fortran standards:

1. Varying-length Character Strings (part 2)
2. Conditional Compilation (fpp), similar to cpp without macros (part 3)

Major New Features of Fortran 200x

Data Abstraction / Object-Oriented Programming::

1. Allocatable arrays as components, dummy arguments, and function results (ISO/IEC Technical Report)
2. Parameterized derived types
3. Derived type I/O
4. Object initialization, constructors and destructors (OOF)
5. Procedure pointers (OOF)
6. (Single) Inheritance (OOF)
7. Polymorphism (OOF)

High Performance Computing / Other Features::

1. IEEE floating point exception handling (ISO/IEC Technical Report)
2. Asynchronous I/O
3. Interoperability with C
4. Internationalization

Minor Technical Enhancements

- Access to command line arguments and environment variables
- Access to status error messages in text form
- Access to standard I/O unit numbers
- Allowing PUBLIC entities to have PRIVATE type
- Derived type encapsulation feature
- Dynamic type allocation
- Enhanced complex constants
- Extended initialization expressions
- Extending min/max intrinsics to CHARACTER
- Generic rate_count in system_clock
- IEEE I/O rounding control
- Increased statement length
- Intent for pointer arguments
- Lower case syntax elements
- Named scratch files
- Passing specific/generic names as arguments
- PUBLIC and PRIVATE derived type components
- Renaming defined operators
- Specifying pointer lower bounds
- Stream I/O
- System clock
- VOLATILE attribute

Features Lacking in Fortran 200x

- specification of operator precedence for user-defined operators
- user-defined exception or event handling
- better interval support, e.g.
  - direct access to operators with directed roundings
  - notation for interval constants (square brackets used for arrays?)
  - flexible optimization control
  - efficient switching of rounding mode
- multiple inheritance
- templates
- generalized module structure
- additional support for HPC

Standardization Problems

- backward compatibility with previous standards
- coexistence of multiple paradigms ("eons")
- too many syntax variants
- extremely tedious and time-consuming corrections/clarifications/ interpretation process
- complexity of language and document
- size and complexity of secondary literature
- size and complexity of compilers
- very little teaching of / training in Fortran
- traditional disinterest of computer scientists
- diminishing (?) user community

But: Is there much choice in Scientific and High Performance Computing ?

Summary

- Fortran 95 is a highly portable, powerful, and efficient programming language for numerical programming and engineering/scientific applications
- Fortran is still the language of choice for many numerical codes, especially in HPC when running on parallel and vector computers
- currently at least 15 Fortran 95 compilers, more Fortran 90 compilers and various source analysis and transformation tools are available
- Fortran compilers usually produce faster code than any other compilers
- new language features greatly improve readability, maintainability, and portability of Fortran 90/95 programs
- computational results continue to be unreliable and incompatible when using various compilers, code optimization and vectorization strategies, and hardware platforms — unless result verification techniques are used
- additional information about Fortran compilers and tools, books and tutorials can be obtained at

`http://www.fortran.com/metcalf.htm`