Scientific Programming 11 (2003) 67–76
IOS Press

Variable precision arithmetic: A Fortran 95 module

J.L. Schonfelder
Computing Services Department, The University of Liverpool, UK
E-mail: j.l.schonfelder@liverpool.ac.uk

Abstract. This paper describes the design and development of a software package supporting variable precision arithmetic as a semantic extension to the Fortran 95 language. The working precision of the arithmetic supported by this package can be dynamically and arbitrarily varied. The facility exploits the data-abstraction capabilities of Fortran 95 and allows the operations to be used elementally with array operands as well as with scalars.
The number system is defined in such a way as to be closed under all of the basic operations of normal arithmetic; no program-terminating numerical exceptions can occur. Precision loss situations like underflow and overflow are handled by defining special value representations that preserve as much of the numeric information as is practical and the operation semantics are defined so that these exceptional values propagate as appropriate to reflect this loss of information.
The number system uses an essentially conventional variable precision floating-point representation. When operations can be performed exactly within the currently-set working precision limit, the excess trailing zero digits are not stored, nor do they take part in future operations. This is both economical in storage and improves efficiency.

1. Introduction

The production of software to provide facilities to support variable and potentially extreme precision arithmetic has been an interest for a number of people for many years [1]. To some extent, this might be said to be a hobby for those who like playing with numbers and computers. However, such facilities can be and have been used in anger to solve real problems where extremes of precision (accuracy substantially greater than available with normal real arithmetic) are needed.

The standard languages of the past, with the exception of Ada and Algol 68, could support these facilities only in the form of procedure libraries. These were in general cumbersome to use; the comparison between programming arithmetic in assembler and a high level language is apt. The MP package by Brent was made somewhat more usable by the production of a pre-processor front-end that allowed more normal arithmetic expressions to be translated into the necessary library calls. The author's Algol 68 package, mlaritha, exploited the capabilities of that language to provide a numeric datatype and operations in the form of an entirely standard-conforming semantic extension, and this was used for some years for the development of high-accuracy approximations to many of the special functions [2]. The advent of Fortran 90 made it possible to provide much of this Algol 68 functionality in a still active language. The feasibility of this was investigated by this author during the design phase of Fortran 90 [3]. A Fortran 90 module providing a proof of concept was subsequently implemented and released via the web [4].

This paper describes the design and development of a more elaborate module that exploits the capabilities of the current, Fortran 95, definition of the language. Fortran 95 is a relatively small extension to the Fortran 90 language [5]. It does, however, add some very important facilities that greatly enhance the ability of the programmer to produce data-abstraction modules that are more effective as semantic extensions. The term "semantic extension" is used here to mean the ability to provide a package that defines a new datatype and the operations to manipulate entities of this type that is so complete that the user of the package can write programs which employ the new datatype in ways that

ISSN 1058-9244/03/$8.00

differ very little from programs manipulating intrinsic datatype objects. Ideally, it should be possible to hide the details of the data representation and the detailed semantics of the operations while at the same time providing all the essential facilities necessary for manipulating objects of this new type that would be needed were the type to be added intrinsically by the core language designers. As currently defined, the language does not enable this ideal to be fully realised but Fortran 95 does allow one to get significantly closer than was possible with Fortran 90.

Fortran 90 provided for the definition of a suitable derived datatype and allowed the overloading of the operations that apply between objects of this type. However, operations on intrinsic numeric types are also defined to apply elementally to conformant arrays of values. That is, as well as the operator plus being applicable between two scalars of type real, say, plus could be applied to real array operands of the same shape. In this case the operation applies to each corresponding pair of real array elements element by element. This elemental functionality could not be expressed in any practicable way in Fortran 90 for derived operations between new user defined types. Fortran 95 corrects this and extends this "elemental" possibility to user-written procedures, including operations. For this to be allowed, however, an elemental procedure must satisfy some fairly tight restrictions. The restrictions are broadly designed to make the procedure verifiably free of side affects.

This is required so as to allow elemental execution in any order or even in parallel. In practice, these restrictions can have some fairly significant ramifications for program design.

Designing a variable-precision number system to have operations with strictly no side effects requires there to be no error exits. The sort of procedure that could implement multiply, say, can only return information to the invoking program via the operation result. For example, a multiply that resulted in some form of overflow could not set a global error flag or cause an error exit. These would constitute a side effect in the terms of the elemental extension rules. In this context, to produce a variable-precision number system where the operations were overloaded both for scalar arguments and were also elementally extended for array arguments, the number system must be closed under the operations that are defined for it. Return values must be defined for all possible values of the arguments. Exceptions must be indicated by suitably defined result values. These must in turn propagate sensibly through the operations if received by them as operands. In designing a variable-precision number representation and operator semantics which satisfy these criteria in reasonably sensible ways, we have tried to define a set of special "exceptional" values and representations that retain as much of the information that can be considered reasonably reliable, given that a numeric exception has occurred somewhere in the chain of operations.

It becomes clear to anyone who works with dynamically controllable floating-point number systems for any length of time that if you have a value exactly representable with a few digits only that is what should be used. It is both wasteful in storage and in processing to retain a possibly large number of redundant trailing zero digits because the current working precision permits such a number. Most variable precision packages normalise results so as to remove both leading and trailing zeros for this reason and this package is no exception.

2. The number system

The datatype chosen to represent numbers with a variable precision and large range is entirely conventional. It employs a fixed radix and uses an integer component to hold the exponent power to that radix. The mantissa is an integer array of variable length; each element holding a "digit". The datatype is defined by

```
TYPE NUMBER
  INTEGER         :: exp=rad+2
    ! holds the base rad exponent
  INTEGER,POINTER :: sig(:)=>NULL()
    ! holds the significand
ENDTYPE NUMBER
```

If `A` is a value in this number representation

```
A = (rad**exp)*SUM(sig(i)*rad**(-i))
  for i=1,N
```

where `rad` is the radix of the number system and `N` is the allocated size of the `sig` array as well as the number of digits stored.

For convenience in converting between externally represented decimal values and such internal numbers, the radix is chosen to be a power of ten. To provide for reasonable efficiency, we want to have a given decimal precision implemented by as few (smallest `N`) digits in the significand as possible. This means we want the highest power of ten representable in an `INTEGER`. To simplify some operations and to enable potentially exact operations to be performed exactly we need also to

have the square root of the radix exactly representable. This further restricts the radix to be an even power of ten. For a typical machine, such as a PC or a Unix workstation where default Fortran integers are 32 bit, the decimal range is 9, i.e. the biggest representable INTEGER is greater than 10^9, but smaller than 10^{10}. This makes a suitable value for rad of 10^8 or 100,000,000.

Normalised numbers have:

```
ABS(sig(i)) < rad    for i=1,N
sig(1)/=0
```

and all `sig(i)` have the same sign, the sign of the value. Digits for `i>N` are either exactly zero or they have been truncated because of the limit set by the current working precision. The system maintains a global limit for the current working accuracy, which determines the maximum number, ndig, of digits to be retained from any operation. This can be changed dynamically at any time.

The value zero is not a properly normalised number in this representation but it is as usual defined as a special case, represented by

```
sig(1)=0, N=1, exp=-rad-2
```

The value of the exponent is strictly irrelevant but it is convenient to use a conventional value in this case for the exponent of zero; we choose therefore an exponent smaller than the smallest allowed normalised exponent, `-rad`.

As with any such floating-point number system, the range of valid exponent values is limited. In performing operations on normalised values, it is possible to produce result values that are too big to be represented, or are too small but not zero. These are the well-known overflow and underflow exceptions. When any of these exceptions occur information is necessarily lost. After an overflow, we no longer know much except that a very large number has been produced. However, we usually do know whether it is a large positive number or a large negative number. Similarly, when underflow occurs we generally know we have produced a very small value of a known sign.

Provided the range of normalised numbers is sufficiently great to handle all reasonable problems, we could essentially treat overflow values as approximations to plus or minus infinity. Similarly, we could treat underflow as approximations to zero. However, again the fact that we usually still know the sign of the very small value, it is better to define underflow as producing either a plus or a minus infinitesimal.

Adding representations coding for these four exceptional cases was the first step taken to attempt to define a number system that was closed under the normal operations of arithmetic. Given that we are working with a large radix, the largest even power of ten representable in an integer, we can arbitrarily define the maximum and minimum exponents as plus and minus radix. This will give a very large range of normalised values. It will sacrifice a relatively limited set of potentially representable values and will allow some of these to be used to represent the "exceptional" values. With the radix `rad=100,000,000` the extreme normalised values are roughly ten to the power plus and minus eight hundred million,

$$(10^8)^{+100000000}, \quad (10^8)^{-100000000}$$

These are very, very large and very, very small numbers. We can conventionally define a representation for values larger than this denoted as `+ovf`, `-ovf`, by

```
sig(1)=+1 or sig(1)=-1, N=1
and exp=rad+1
```

We can also define underflowing small values, denoted by `+unf`, `-unf`, to be represented

```
sig(1)=+1 or sig(1)=-1, N=1
and exp=-rad-1
```

By definition, these are specific values that are either larger or smaller than any normalised values. We can define the semantics of the operations so that overflow and underflow result in one or other of these values being returned.

Unfortunately, once we start trying to define sensible semantics for the arithmetic operations when these special values are included within the value set for the operands, we find this is not sufficient. We have some operations that do not have a defined result of any magnitude. For example, the operation exact-zero divided by exact-zero, `0/0`, is intrinsically ill determined. Similarly, the operation `ovf-ovf` could produce any value whatsoever. It is therefore effectively indeterminate. We must be able to represent the result of such operations. In practice we have not found any need to distinguish between these two cases. The same special value representation can be used for both situations. A suitable representation for the "indeterminate" value, `ind`, could be

```
sig=>NULL() and exp=rad+2
```

which is also the representation we have chosen for the initial value for any object of the datatype; the significand is disassociated and the exponent is larger even than that of the overflow value.

When we look more closely at defining how these additional special values should propagate through the arithmetic operations, we find we need further special values to obtain a sensible closed number system that preserves as much information as possible. We can also generate values whose magnitude is unknown but where the sign is predictable. The operation of `-ovf*unf`, the product of a very large negative value by a very small positive value, must be negative but otherwise we have no way of determining the magnitude. We therefore need representations for, `-unk` and `+unk`, negative and positive unknown. Here,

```
sig(i)=-1 or sig(i)=+1, N=1
and exp=rad+2
```

would be suitable representations. The complete set of values in the number system is therefore

```
-unk, -ovf, -x, -unf, 0,
+unf, +x, +ovf, +unk, ind
```

where `x` denotes a properly normalised value.

There is a superficial similarity between some of these special values and those added to the set of normal numbers defined by the IEEE floating point standard [6]. The values `+ovf`, `-ovf`, and ind are similar to IEEE $+\infty$, $-\infty$ and Nan. However, the IEEE special values do not include analogues of the signed unknowns, $\pm$`unk`, or the signed underflow, $\pm$`unf`. The IEEE arithmetic deals with underflow by first having denormalised values for "gradual underflow" and signed zeros which continue to carry sign information when eventual underflow "flushes to zero". With a variable precision system such as here gradual underflow via denormalised values is not a practical option and since the Fortran standard does not permit a zero integer to be signed are underflow values needed to provide closure. The IEEE special values were designed not so much to provide a closed number system as to form part of a system for handling arithmetic exceptions. Fortran 95 does not define an exception handling system so the special value set defined here is intended to provide a closed number system that will preserve and propagate as much numeric information as reasonable through a calculation sequence even where no exception flagging or handling facility exists.

3. The arithmetic operations

The algorithms chosen to implement the arithmetic operations are all entirely conventional. Addition and subtraction use a working register at least two digits bigger than that required by the working precision. The operand with the largest exponent is copied into the appropriate place in this register and the smaller operand added or subtracted depending on the relevant signs. The resulting register value is then normalised to the above conventions, removing any trailing zeros. The multiply and divide algorithms are variations on the "school" methods for "long-multiplication" and "long-division".

A key issue is how the special case values should propagate if received as values for operands. The following table gives a definition of the semantics for the operation plus (`+`) for operands drawn from the extended set defined above.

The entry `Alg` denotes the application of the basic plus algorithm to normalised operands. Of course, it is possible that this algorithm may produce a result that is zero, overflows ($\pm$`ovf`) or underflows ($\pm$`unf`), as well as properly normalised values.

The definitions chosen for the special-case operations need some discussion.

Obviously any operation involving an `ind` operand, which indicates a total lack of information about the value, must produce an `ind` result. As we have no information at all about the indeterminate value, we can have no information about the result value.

For operations involving a signed but otherwise unknown value, we can return a signed unknown result if we can guarantee that the sign remains predictable. Otherwise, we must return an indeterminate value.

Since the only information we have about an overflow value is that it is very large, the operation of `ovf-ovf` has no predictable value or sign. We therefore define the result here as `ind`. Adding an overflow to any other value we assume will continue to produce a similar signed overflow. The only case where this could be seriously wrong would be `ovf-x`. If the `ovf` was the result of a marginal overflow and `x` is also very close to the overflow threshold, the true result could well be a reasonably small value and not at all close to an overflow at all. We do know the sign of the overflow must be preserved. We perhaps strictly should return a signed unknown in such cases. However, in most cases the normalised value will be of reasonable magnitude so the result is likely to be large. The best, or perhaps least worst, value to propagate in such an operation is an `ovf`.

In all cases of addition operations involving the underflow ($\pm$`unf`) values with finite or large values, we treat the underflow values as if they were zero. Again,

Exceptional value propagation by addition operation A + B

B A	-unk	-ovf	-X	-unf	0.0	+unf	+X	+ovf	+unk	ind
-unk	-unk	-ovf	-unk	-unk	-unk	ind	Ind	ind	ind	ind
-ovf	-ovf	-ovf	-ovf	-ovf	-ovf	-ovf	-ovf	ind	ind	ind
-X	-unk	-ovf	Alg	-X	-X	-X	Alg	+ovf	ind	ind
-unf	-unk	-ovf	-X	-unf	-unf	0.0	+X	+ovf	ind	ind
0.0	-unk	-ovf	-X	-unf	0.0	+unf	+X	+ovf	+unk	ind
+unf	ind	-ovf	-X	0.0	+unf	+unf	+X	+ovf	+unk	ind
+X	ind	-ovf	Alg	+X	+X	+X	Alg	+ovf	+unk	ind
+ovf	ind	ind	+ovf	+ovf	+ovf	+ovf	+ovf	+ovf	+ovf	ind
+unk	ind	ind	ind	ind	+unk	+unk	+unk	+ovf	+unk	ind
ind	ind	ind	ind	ind	ind	ind	Ind	ind	ind	ind

this could be somewhat erroneous for an operation like `x-unf` if the true values are close to the thresholds. However, in the vast majority of operations this will not be the case. The most useful way of propagating the effect of the exception in such cases is to treat underflow like zero. However, it is well known that for normal fixed-precision floating-point arithmetic some algorithms are subject to serious loss of accuracy if underflow is automatically set to zero. Some care would therefore be needed in coding such problems using this number system. The lack of an exception-flagging system makes this even trickier.

The remaining exceptional propagation is `unf-unf`; this must produce an underflowing value but one that has an indeterminate sign. In a sense this is an approximation to zero, and as we are not distinguishing between exact and approximate values, a reasonable result is zero.

The algorithm used to perform the addition operation for normalised operands is designed to produce a result with the maximum potential precision, subject to the constraint that no more significant digits will be retained than are permitted by the current working precision setting. If both operands and the "exact" result can be accommodated within the current working precision, an "exact" result will be returned with only significant digits included; neither leading nor trailing zero digits will be retained. If the number of potentially correct digits is greater than the number required by the current working precision, the result is truncated to this length.[1]

[1] The radix is large. Each digit corresponds to a number of decimal digits. For a typical system using IEEE arithmetic where default integers are 32-bit this number is eight. However, the first digit may contain only one decimal digit. Thus to guarantee a given decimal accuracy, nD say, the number of digits retained must be at least CEILING(n/8) + 1. For a particular working precision, the actual number of decimal digits stored for any given value can vary by 8D. There is a trade-off between fine granularity precision control, that needs a small radix, and efficiency that requires a large radix.

The definitions for exception value behaviour and the operation of the subtraction algorithm follow directly from the above.

The definitions that have been used for multiplication are shown by the table below. Zero is treated as an exact value even though it may have arisen merely as an approximation. In this number system as in most conventional floating-point systems, we do not distinguish these cases. Therefore, if either operand is a zero the result is returned as zero. This is the case even when the other operand is indeterminate. In most cases an indeterminate value will merely be a value of unknown sign and unknown magnitude. For all other cases when either operand is indeterminate the only possible result is indeterminate. If either operand is a signed unknown ($\pm$`unk`), the result is also a signed unknown. The product of an overflow value with another overflow or a normalised value should result in an overflow result of the appropriate sign. Of course, there will be occasions when this can produce rather seriously wrong results, For example, a product of a nearly underflowing normalised value with a barely overflowing one could produce a finite value of almost any magnitude. However, on the assumption that most normalised values will be of relatively finite range and all we know about an overflow is that it is very large, propagating the overflow appears to be sensible. An operation such as ($\pm$`ovf`)*($\pm$`unf`) has no predictable magnitude but a well-determined sign and the appropriate signed unknown is the required result.

The product of normalised values, indicated by `Alg` for algorithm, uses a version of the common schoolroom long-multiplication algorithm. Each digit is multiplied by each other digit and the result accumulated into a register the size of which is `na+nb+1`, where na and nb are the sizes of the argument significands. The multiplication and accumulation process is done in such a way that the accumulator is normalised, modulo rad, at all times. In order to ensure that no integer overflow occurs while multiplying two digits that could be

Exceptional value propagation by multiplication operation A * B

B A	-unk	-ovf	-X	-unf	0.0	+unf	+X	+ovf	+unk	ind
-unk	+unk	+unk	+unk	+unk	0.0	-unk	-unk	-unk	-unk	ind
-ovf	+unk	+ovf	+ovf	+unk	0.0	-unk	-ovf	-ovf	-unk	ind
-X	+unk	+ovf	Alg	+unf	0.0	-unf	Alg	-ovf	-unk	ind
-unf	+unk	+unk	+unf	+unf	0.0	-unf	-unf	-unk	-unk	ind
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
+unf	-unk	-unk	-unf	-unf	0.0	+unf	+unf	+unk	+unk	Ind
+X	-unk	-ovf	Alg	-unf	0.0	+unf	Alg	+ovf	+unk	Ind
+ovf	-unk	-ovf	-ovf	-unk	0.0	+unk	+ovf	+ovf	+unk	Ind
+unk	-unk	-unk	-unk	-unk	0.0	+unk	+unk	+unk	+unk	Ind
ind	ind	ind	ind	ind	0.0	ind	ind	ind	ind	ind

as large as `rad-1`, the digits are decomposed modulo `SQRT(rad)`, and the multiply and accumulation done using these components.

If the operands are exact and the current working precision is such that the result can be accommodated without truncation, the result remains exact. Otherwise the result will be truncated to the current working precision.

Division is defined to propagate the exception values in an entirely analogous way. The algorithm is also an analogue of the schoolroom long division. Real arithmetic is used to make an estimate of the first digit in the quotient and a new dividend produced by multiplication and subtraction. This makes use of the same modulo `SQRT(rad)` decomposition and accumulate procedure as for multiplication so as to ensure normalisation at all times without the risk of integer overflow. Division of any value by zero results in an indeterminate value, `ind`, being returned.

4. Logical comparisons

The treatment of the logical relations in the presence of these exception values needs some thought. In reality such a number system needs a trivalent logic to properly express comparisons. What is the result of an `ind<ovf` comparison? Since `ind` represents an indeterminate value of unknown sign or magnitude the answer could be either true or false, perhaps maybe! However, overloads of the "less-than" operator must return true or false – there is no available representation in the `LOGICAL` datatype value set for "maybe". Therefore we need to define a conventional set of responses. What we have done is to simply use the specific representations and performed the comparisons as if these were normalised values. We have also said that since an indeterminate value is basically an inexact zero (unknown sign and very high uncertainty of magnitude) for logical comparisons we will treat these also as zero. The signed exception values are compared strictly according to their representation. This makes `+unk > +ovf > +x >+unf`, etc. The following table shows the detailed results for the operation `<`.

It should be noted that we need elemental overloads of these logical relation operators producing simple LOGICAL results since such array comparisons are likely to be used to produce mask arrays for WHERE statements or other similar array contexts. Again in the absence of an exception handling system we must define valid outcomes for all possible operand values, no matter how strange the result.

Similar tables are available for each of the other relational operations and these are included in the module documentation [7].

5. Additional facilities

A small number of the basic relevant intrinsic functions are also provided as generic elemental overloads. The functions, `ABS` and `SIGN` extend the obvious operations to `NUMBER` values.

Input conversions are provided by one of two generic versions of a function `NUM`. If invoked with an `INTEGER` argument the equivalent `NUMBER` value is returned. If invoked with a character string argument where the string denotes a real value in either fixed- or floating-point form, the equivalent `NUMBER` value is returned.

Output is performed by three procedures, `CHAR`, `EFCHAR`, and `FFCHAR`. An overload for the `CHAR` function for a `NUMBER` argument produces a character string denoting a floating-point real. This function performs an exact conversion regardless of the current working precision. The other two functions produce character strings denoting the NUMBER value, the first in E-format and the second in F-format; the width and number of decimal places retained are provided as additional arguments.

Exceptional value comparison by relational operation <

A<B	-unk	-ovf	-6.86	-unf	0.0	+unf	+2.50	+ovf	+unk	ind
-unk	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
-ovf	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
-6.86	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
-unf	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
0.0	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE
+unf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE
+2.50	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
+ovf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
+unk	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
ind	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE

A version of the `INT` function converts from a `NUMBER` value to the equivalent `INTEGER` by truncation toward zero.

The base module does not provide overloads for the elementary mathematical functions such as `SQRT`, `LOG`, `EXP`, etc.. These are being provided via a separate dependent module that is under development [8]. This base module plus additional facilities provided by separate but dependent modules was a deliberate design. Since, it is very much easier to produce efficient dependent modules if these can access the details of the number representation directly the component structure of the datatype and the parameters of the representation are all `PUBLIC`. Fortran 95 only supports two levels of access control, fully private to the module or public to all using programs. It would be useful in situations like this to have an intermediate restricted access which would make representation details visible to dependent modules but hidden from normal user programs.

6. Performance

The performance of the package is difficult to predict in detail because of the large number of factors involved. The performance of the various operations is in general dependent on the current working precision. However, if operations are performed with operands that do not require the full length of storage, the algorithms used will automatically adjust. The time taken to perform an operation will depend on the actual stored length of the operands as well as the working precision. In the case of addition and subtraction it will also depend to some extent on the relative magnitudes of the operands; if the two operands have very different exponents the "shift" to align the significands will possibly reduce the number of digit sums that need to be done. Nevertheless, for a large number of uses the package will be employed with primarily "full-length" values. Timing tests have been done using such full-length operands of commensurate magnitudes. The results are presented as functions of the working precision. These show the expected theoretical behaviour. The addition/subtraction performance is essentially linear in its dependence on precision, and multiplication and division are quadratic.

The tests consisted of timing two loops that differ only by the second including an additional operation of the sort being tested. An array of possible operands is constructed with the digits chosen as random integers less than `rad` in magnitude. Each number so constructed is of full length but all have the same exponent. The control loop generates a random index, which is used to select the operand and this is assigned to another array element, also selected at random. This stops any clever optimiser avoiding actually executing the loop. The main timing loop is identical to the control loop except that the assignment now includes an additional execution of the chosen operation. The essential code fragment for a test of multiplication is shown in Fig. 1.

These loops are repeated for a number of different precisions. The tests were run on a number of different PCs with different clock speeds but using the same NA-Software Fplus compiler. The results normalised for clock speed are quite reproducible. Timings were done for samples of sufficient size to provide approximately 10% accuracy for the measured operation times. In units of "clock-ticks" the performance measured was, for addition/subtraction,

```
Cost per operation = 760 + 7.4 *acc
```

for multiplication

```
Cost per operation = 760 + 75*acc
+ 3.9*acc**2
```

for division

```
Cost per operation = 1800 + 110*acc
+ 4.4*acc**2
```

```
type(NUMBER) :: A(20),B(20),C

. . .
! fill A and C with full precision random values
. . .

  call TIMER(t1)
  DO j=1,count
    k=IRAND(1,20)
    m=IRAND(1,20)
    B(k)=A(m)      ! simple assignment
  ENDDO
  call TIMER(t2)
  td = t2-t1
  call TIMER(t1)
  DO j=1,count
    k=IRAND(1,20)
    m=IRAND(1,20)
    B(k)=A(m)*C    ! assignment with extra operation
  ENDDO
  call TIMER(t2)
  tp=t2-t1
  t=(tp-td)/count   ! time in seconds of adding one extra * operation
```

Fig. 1.

where acc is the current working precision in decimal digits. This means that on a 400 MHz PC at 200D one multiplication takes approximately 430 microseconds.

It should be noted that although the precise values of the coefficients in such timings are quite reproducible over a range of clock speeds, they are likely to be somewhat compiler dependent. A processor system that does not support garbage collection is likely to have unreliable performance since long run times could be caused by memory problems. However, the form of precision dependence is entirely a function of the algorithms used.

If both operands are "exact", the time taken to perform the operation is going to depend on the effective lengths of the operands, na and nb, and separately on the current precision. In fact, if the result is also exact within the current precision, the current precision will be all but irrelevant. In particular, in the important special case of multiplication by a simple integer the dependence automatically reverts to being linear in current precision.

It should be noted that for this representation for NUMBER values the package will leak memory. As a result it can cause problems on a system that does not support garbage collection. This problem virtually disappears on a system supporting the proposed Fortran 2000 extension that allows allocatable components. In this case the pointer significand component is replaced by an allocatable array component, and there are some minor consequent changes to the algorithm code. The package has an almost identical user interface in this version and, as well as being more robust in its memory management it is somewhat more efficient in execution.

7. Example use

An example of the use of the module is shown below. This merely employs the scalar versions of the operators and is neither the most sophisticated of algorithms nor a particularly robust or efficient implementation. It does illustrate how easy it is to work with variable precision quantities using a facility of this sort. The example is the classic one of calculating π to an arbitrary user-selected number of decimal places. The value of π is calculated using the well-known Machin identity,

$$\pi/4 = 4\mathtt{arctan}(1/5) - \mathtt{arctan}(1/239)$$

```
PROGRAM EXAMPLE_PI
! calculates PI to a user requested number of decimal digits using
! the Machin identity PI/4 = 4 arctan(1/5) - arctan(1/239)
! the arctan values are obtained by summing the Taylor series
  USE VARIABLE_PRECISION_ARITHMETIC
  IMPLICIT NONE
  type(NUMBER) :: pi,atb5,atb239,mzsq,rn
  INTEGER :: acc,n
  WRITE(*,ADVANCE='NO',FMT='(A)')"Input the desired number of digits for PI?"
  READ(*,FMT='(I10)') acc
  n=PRECISION(acc) ! working precision set to provide
                   ! at least acc decimal digits
  ! calculate arctan(1/5)
  rn=NUM("0.2")
  mzsq=NUM("-0.04")
  atb5=rn
  n=1
  DO ! until sum converges
    rn = (NUM(2*n-1)*mzsq)*rn/NUM(2*n+1)
    IF(atb5%exp-rn%exp > ndig) EXIT ! remaining terms too small to
                                    ! effect the sum
    atb5=atb5 + rn
    n=n+1
  ENDDO
  ! calculate arctan(1/239)
  rn=NUM(1)/NUM(239)  atb239=rn
  n=1
  DO ! until sum converges
    rn = NUM(2*n-1)*rn/NUM(-57121*(2*n+1)) ! 57121= 239*239
    IF(atb239%exp-rn%exp > ndig) EXIT ! remaining terms too small to effect
                                      ! the sum
    atb239=atb239 + rn
    n=n+1
  ENDDO
  pi=NUM(16)*atb5 - NUM(4)*atb239
  WRITE(*,FMT='(A)') FFCHAR(pi,acc+5,acc-1)
END PROGRAM EXAMPLE_PI
```

Fig. 2.

The arctan values can be conveniently calculated to any desired accuracy by summing the necessary number of terms in the Taylor series

$$\mathtt{arctan(z)} = \sum_{\mathtt{n=0}}^{\mathtt{N}} \mathtt{R_n} + \mathtt{E}$$

where

$$\mathtt{R_n} = (-1)^{\mathtt{n}}\mathtt{z}^{2\mathtt{n}+1}/2\mathtt{n}+1$$

and

$$\mathtt{E} \leqslant |\mathtt{R}_{\mathtt{N}+1}|$$

The code implementing this example is shown in Fig. 2.

It should be noted that the output is quite simple and will cause a buffer overflow on most systems if the accuracy requested is too large. The `FFCHAR` procedure produces a result that is the value expressed as a character string in `Fw.d` format where `w=acc+5` and `d=acc-1`. A more complex and robust method of output could be produced but this would only make for a more obscure example.

On a 400 MHz PC this program produces 1000 digits in 5 seconds and 2000 in 25 seconds, further indicating that the algorithms are dominated by the times for multiply/divide which are O(`ndig`^{2}).

References

[1] R.P. Brent, A Fortran multiple-precision package, *ACM Trans. Math. Software* **4** (1978), 57–70.
J.L. Schonfelder and J.T. Thomason, Applications support by direct language extension, in: *International Computing Symposium*, Gelenbe and Potier, eds, North Holland, 1975.
J.L. Schonfelder, Arbitrary precision arithmetic in Algol 68, *Software – Practice and Experience* **9** (1979), 173–182.
D.M. Smith, A Fortran Package for floating-point multiple precision arithmetic, *ACM Trans. Math. Software* **17** (1991), 273–283.

[2] J.L. Schonfelder, The production of special function routines for a multi-machine library, *Software – Practice and Experience* **6** (1976), 71–82.
J.L. Schonfelder, Chebyshev expansions for the error and related functions, *Math. Comp.* **32** (1978), 1232–1240.
J.L. Schonfelder, Very high accuracy Chebyshev expansions for the basic trigonometric functions, *Math. Comp.* **34** (1980), 237–244.

[3] J.L. Schonfelder, Semantic extension possibilities in the proposed new Fortran, *Software – Practice and Experience* **19** (1989), 529–551.
[4] An early version of the module in strict Fortran 90 was released via the web and at the time of writing it can still be found at http://www.pcweb.liv.ac.uk/jls/vpa11.htm.
[5] ISO/IEC 1539: 1991, Information technology – Programming Languages – Fortran and ISO/IEC 1539-1: 1997, Information technology – Programming Languages – Fortran Known as Fortran 90 and Fortran 95, respectively.
[6] IEEE 754: 1985 or IEC 559: 1989 Binary Floating-point arithmetic for microprocessor systems.
[7] The module described here and its user documentation is to be found at the time of writing at http://www.pcweb.liv.ac.uk/jls/vpa20.htm.
[8] B.G.S. Doman has produced a set of dependent modules providing near-optimal approximations for the basic elementary functions. This work will be published elsewhere. Both packages are to be bundled and distributed with the Fplus compiler from NASoftware, delves@nasoftware.ac.uk.