EXPERIENCE WITH A HIGH LEVEL LANGUAGE THAT SUPPORTS INTERVAL ARITHMETIC

R. Morrison, A. J. Cole, P. J. Bailey, M. A. Wolfe & M. Shearer

University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX.

Abstract

An extension of the language S-algol[4] called Triplex[5] which facilitates the use of interval arithmetic and which is similar to triplex algol 60[3] is described. Experience in the use of Triplex is reported. In particular, a Triplex program corresponding to a triplex algol 60 program of Nickel[19] is given, together with numerical results.

Introduction

Algorithms which contain non-trivial amounts of numerical calculation cannot usually be implemented exactly using the high speed floating point unit of a computer for the following reasons.

(i) Data-values may be known only approximately.
(ii) Real numbers are, in general, represented only approximately by machine numbers, giving rise to input-output conversion errors.
(iii) The arithmetic operations which are defined on the set of machine numbers are themselves inexact representations of the corresponding arithmetic operations which are defined on the set of real numbers.
(iv) Infinite sequences of arithmetic operations which may be required by a given algorithm must necessarily be truncated after a finite number of operations have been performed.

Furthermore, if an algorithm for solving a properly-posed problem in numerical mathematics is supported by a satisfactory theoretical analysis, then there will exist theorems which contain sufficient conditions for the existence and uniqueness of a solution, and for the convergence or termination of the algorithm. It is usually rather tedious to verify by hand that these conditions hold and it is desirable that they should be verified by using the computer. Unfortunately, many conditions are of the form $a \leq b$ where a and b are real numbers which must, in practice, be estimated by making often long (and inexact) numerical calculations. Clearly it is impossible to decide rigorously whether or not $a \leq b$ using ordinary machine arithmetic. The paper by Rall[20] contains an interesting example.

The preceding remarks reveal two serious difficulties in numerical computing, namely that it is usually impossible to implement a numerical algorithm exactly on a computer, if ordinary machine arithmetic is used, and also that it is usually impossible to decide, again by using ordinary machine arithmetic only, whether or not sufficient conditions for the applicability of the algorithm hold. Both of these difficulties have been and are being satisfactorily resolved by using interval mathematics[15,18,9].

High level language support for triples

Interval arithmetic has been incorporated into several high level languages. Yoche[24] has written a package of interval arithmetic Fortran sub-routines which is designed for use with a pre-compiler in order to allow INTERVAL to be used as a standard data type. More recently, Guenther & Marquardt[7] have implemented interval arithmetic in Algol 68, introducing a data type **intval** to represent real intervals. The original implementation of the interval arithmetic in a high level language appears to be due to Apostolatos et al[3] who extended Algol 60 by introducing the data type **triplex**. A triplex number x is an ordered triple [inf(x), main(x), sup(x)] of real numbers such that $\inf(x) \leq \mathrm{main}(x) \leq \sup(x)$.

In this paper we describe the facilities for triple arithmetic which are embedded in the language S-algol[4,17]. The implementation issues of the new language called Triplex are described by Cole & Morrison[5]. The advantage of the embedding approach is that the triple can be a separate data object with language constructs tailored to manipulating that data object. This means that programs which use triples will be considerably shorter, with all the attendant benefits, than programs which use subroutine packages for example. Another advantage is that the system can guarantee the integrity of the triples without fear that the user may accidentally alter one of the triples. The only disadvantage of this approach is that the compiler may have to be altered to achieve the embedding. Fortunately this is not a serious problem in this case.

Language extension is normally achieved in two ways - syntactic extension and procedural extension. The advantage of syntactic extension is that since the extension may include infix operators the programs will be shorter and clearer. The disadvantage is that it is more difficult to achieve syntactic extension than procedural extension and that the designer often runs out of

CH1892-9/83/0000/0074$01.00

suitable symbols in the character set. On the other hand, while procedural extension is easier to implement it is often less safe and it is clumsier for the user. The approach taken in Triplex is to mix the two types of extension to give a balance and the advantages of both.

Triplex is based on the language S-algol which itself is designed using two semantic rules attributable to Strachey[21] and Landin[13]. The first rule is the principle of correspondence which states that the rules governing the use of names in a language should be the same for all names. This means that if names for triples are introduced then it must be done according to the rules of the language that already exist in order to maintain the orthogonality of the language. In particular, Tennent[22] points out that there must be a one to one correspondence between the methods of introducing names in declarations and as parameters.

The second rule is the principle of data type completeness which states that all data objects have the same 'civil rights' in the language. Thus if a vector of reals is allowed then so should vectors of any type including triples. This, of course, does not mean that all operators need to be defined on all data types since operators are merely syntactic sugarings of functions, but rather that the rules for combining data objects are complete with no gaps.

S-algol is complete under the principles of correspondence and data type completeness and in order to retain the language orthogonality so should the Triplex extensions be. This means that there should be names of type triple, triple expressions, vectors of triples, triples as fields of structures, triples as parameters of procedures and procedures whose result is a triple. In fact the language Triplex provides the user with a method of defining triples that ensures they are well formed, infix operators for triple arithmetic, standard functions normally defined for real numbers and a guarantee that the triples are protected from misuse.

Interval extension

In order to guarantee that the triples are well formed we must know a little about the arithmetic performed by the host computer. The reasoning is as follows. In any implementation, let M be the set of machine numbers such that $L = x_0 < x_1 \ldots\ldots\ldots\ldots < x_{n-1} < x_n < U$. If x is an x_i of M then x^- denotes x_{i-1} provided $x \neq L$ and x^+ denotes x_{i+1} provided $x \neq U$. The set of machine triples T is defined by

$$T = \{ [\ i,m,s\ ] \mid i,m,s \quad M \text{ and } i \le m \le s \}$$

More often than not the real number we wish to use is not in M. Let x' be a real number not in M which lies between the values x and x^+ which are in M. Then x' may be represented by the triples

$$[\ x,x,x^+\ ] \text{ or } [\ x,x^+,x^+\ ]$$

The choice of x or x^+ for the main part is not usually important. If it is, calculations should be made to a greater accuracy than the end points and some measure of distance to the end points used to choose between x and x^+.

More important is how to find x and x^+ on machines that are computing to their greatest accuracy. x^+ may be found from x by adding a one to the low order bit of the mantissa of x except when x = U. Similarly x^- can be found by subtraction except when x = L.

On machines which truncate the result of real arithmetic, x' will be truncated to x if x' is positive and x^+ if x' is negative. In either case x or x^+ can be calculated from the other as outlined above. On machines which round away from zero, i.e. increase the absolute magnitude of the number, x' will be rounded to x^+ if positive and x if negative. Finally machines which round randomly may give either x or x^+. To be sure that the number y given by such a machine is represented by the correct interval $[y^-,y^+]$ must be used. This technique may be more generous than necessary when x' is in M. However, this is both rare and usually difficult to spot automatically.

Triplex language facilities

A triple may be formed using the syntactic rule

```
<triple-expression> ::= [<real-expression>,
                         <real-expression>,
                         <real-expression>]
```

Thus

```
[ 1.3,1.3,1.3 ]
```

forms the triple equivalent of the real number 1.3. When a triple [inf,main,sup] is formed, a check is made to ensure the relation

$$\text{inf} \le \text{main} \le \text{sup}$$

holds. The interval is then extended as described above. Once formed the values in the triple can be inspected but never altered.

As with all other data objects in the language a triple may be declared. The syntactic rule is

```
<triple-declaration> ::= let <identifier>
                     [=|:=] <triple-expression>
```

Thus the above triple could be given a name by

```
let one.three = [ 1.3,1.3,1.3 ]
```

and the name used anywhere a triple expression is valid. As an aside if the symbol '=' is used in the declaration the name is constant and may not be assigned to i.e. given a different triple value. If the symbol ':=' is used in the declaration then the name is variable. Thus the only difference between a constant and a variable is that a constant may not be updated. This is guaranteed by the compiler. This type of constant is called a **dynamic constant**[8] and should not be confused with the manifest constants of, say, Pascal[23] since the initialising value can be any legal expression in the language and not just literals and, moreover, the concept also applies to vectors and structures.

There are four arithmetic operations defined on triples. The syntactic rule is

<triple-expression> ::= <triple-expression>
[+|-|*|/]<triple-expression>

Taken along with the other rules for expression formation in the language, expressions of arbitrary complexity may be formed.

For the user's convenience the compiler will coerce integers and reals to triples when necessary. For example

2 * [1.3,1.3,1.3]

gives the same result as

[2,2,2] * [1.3,1.3,1.3]

Along with the arithmetic operations, two relational operations are defined on triples. They are equal and not equal denoted by the symbols = and ≠ respectively.

Let T1 = [i1,m1,s1]

and T2 = [i2,m2,s2]

then T1 is equal to T2 if and only if

i1 = i2 and m1 = m2 and s1 = s2

Some implementations of triples or intervals[3] allow other relational operations such as less than, greater than etc. While these operations may be well defined, such definitions are not universally acceptable and therefore constitute a pit for the unwary to fall into in the case of overlapping intervals. In our opinion it is too dangerous to include these operations, especially since they may be written as functions if required. It is then the user's responsibility to take care when using them and hopefully to fully understand the consequent implications.

A triple may be input by using the standard function readt which has the following specification

procedure readt(**file** t → **triple**)

This performs in a similar manner to **readi** or **readr** and causes three real literals to be read from the relevant file. These literals may be preceded by any combination of newline, space or horizontal tab characters.

Output of a triple is provided by means of the standard function tformat having the following specification

procedure tformat(**triple** t;**int** b,a → **string**)

where 't' is the triple to output, 'b' is an integer specifying the number of digits before the decimal point and 'a' another integer specifying the number of digits after the decimal point. The result is a string which may be subsequently written out using a **write** clause. The string consists of three real numbers, in the format specified, separated by a single space, in the order from left to right of **inf,main,sup.**

It should be noted that in order to maintain the highest degree of accuracy on input and output, the machine arithmetic for triple I/O is simulated. This is naturally quite time consuming.

There are a number of standard functions of which the following are the most commonly used

inf main sup abs sqrt ln exp atan sin cos trip

The arithmetic functions are the Triplex equivalent of the corresponding real functions whereas inf, main and sup take a triple parameter and return the real part of the triple corresponding to their name. The function trip may be used to override the automatic triple extension. For example

[1,1,1]

needs no extension since the integers are represented exactly in the computer. Therefore the user can take advantage of this by using

trip(1)

which will form the triple with no extension.

The user may also define triple functions and the following example is in fact the function abs written in Triplex for triple arithmetic.

```
procedure tabs( ctriple t → triple )
case true of
int( t ) ≥ 0 : t
sup( t ) ≤ 0 : [-sup( t ), -main( t ),-inf( t )]
default : [ 0,rabs( main( t ) ),
           if rabs( inf( t ) ) ≥ sup( t )
           then rabs( inf( t ) )
           else sup( t ) ]
```

Triple arithmetic

Before we describe a solution to a numerical problem using the Triplex language we will define the arithmetic operations +, -, * and / corresponding to addition, subtraction, multiplication and division respectively on triples. They are based on the definitions of these operations on intervals given by Good and London[6]. They are as follows.

Let

T1 = [i1,m1,s1]
T2 = [i2,m2,s2]

Addition

T1 + T2 = [i1 + i2,m1 + m2,s1 + s2]
provided L ≤ i1 + i2 and s1 + s2 ≤ U

Subtraction

T1 - T2 = [i1 - s2,m1 - m2,s1 - i2]
provided L ≤ i1 - s2 and s1 - i2 ≤ U

Multiplication

The main product T1 * T2 is always m1 * m2 but the values for inf and sup is determined by the sign analysis in the table below

	T1 [i1,s1]	T2 [i2,s2]	T1 * T2
1	[≥0,≥0]	[≥0,≥0]	[i1*i2,s1*s2]
2	[≥0,≥0]	[<0,≥0]	[s1*i2,s1*s2]
3	[≥0,≥0]	[<0,<0]	[s1*i2,i1*s2]
4	[<0,≥0]	[≥0,≥0]	[i1*s2,s1*i2]
5	[<0,≥0]	[<0,≥0]	[min(i1*s2,s1*i2, max(i1*i2,s1*s2)]
6	[<0,≥0]	[<0,<0]	[s1*i2,i1*i2]

7	[<0,<0]	[≥0,≥0]	[i1*s2,s1*i2]
8	[<0,<0]	[<0,≥0]	[i1*s2,i1*i2]
9	[<0,<0]	[<0,<0]	[s1*s2,i1,i2]

provided L ≤ min(i1*i2, i1*s1, i1*s2, s1*s2) and max(i1*i2, i1*s1, i1*s2, s1*s2) ≤ U

Division

The quotient is undefined if the divisor spans zero. When it does not the main value is always m1/m2. The table below defines the operation.

	T1	T2	
	[i1,s1]	[i2,s2]	T1 / T2
1	[≥0,≥0]	[>0,>0]	[i1/s2,s1/i2]
2	[≥0,≥0]	[<0,<0]	[s1/s2,i1/i2]
3	[<0,≥0]	[>0,>0]	[i1/i2,s1/i2]
4	[<0,≥0]	[<0,<0]	[s1/s2,i1/s2]
5	[<0,<0]	[>0,>0]	[i1/i2,s1/s2]
6	[<0,<0]	[<0,<0]	[s1/i2,i1/s2]

Computational experience using triplex

Several algorithms for the solution of non-linear algebraic equations and for optimization have been implemented in Triplex. Among these are the algorithms of Moore[16], Madsen[14], Alefeld and Herzberger[2], Krawczyk[12], Alefeld[1] and Hansen[10,11]. Listings of some of the Triplex programs corresponding to these algorithms may be obtained from the authors of this article.

An illustration of how Triplex S-algol may be used is provided by the implementation of the interval Newton algorithm for bounding a zero of f:R→R in triplex-Algol 60 which is contained in section 7.5 of the article by Nickel[19]. Nickel's program bounds the zero x* = 0 of f:R→R defined by

$$f(x) = x/(1+|x|)$$

in the triple x = [-7,4711,247921].

The following Triplex S-algol program also implements the interval Newton algorithm given by Nickel.

```
procedure f( triple x → triple )
x / ( trip( 1 ) + tabs( x ) )

procedure f.prime( triple x → triple )
begin
     let M = 1 / ( ( 1 + 1e6 ) * ( 1 + 1e6 ) )
     let z = trip( 1 ) / tpower( trip( 1 ) +
            tabs( x ),2 )
     let z.i := inf( z ) ; let z.m := main( z )
     let z.s := sup( z )
     if z.s > 1 do z.s := 1
     if z.m > 1 do z.m := 1
     if z.i > M do z.i := M
     if z.m > M do z.m := M
     [ z.i,z.m,z.s ]
end ! f.prime

! MAIN PROGRAM

let n := 0 ; let n.max = 16
let x := [ -7,4711,247921 ]
write "n":3,"inf(x)":11,"main(x)":25,
      "sup(x)":23,"'n"

while

while n ≤ n.max do
begin
     write n:3,tformat( x,0,16 ),"'n"
     let y = trip( main( x ) )
     let z = y - f( y ) / f.prime( x )
     let I = tintsct( z,x )
     if I( tintsct2 ) do
     begin
          write "'nThere is no zero of f in x."
          abort
     end
     let z.x.i = I( tintsct1,1 )
     let z.x.s = I( tintsct1,2 )
     let z.m = main( z )
     if z.x.i ≤ inf( x ) and sup( x ) ≤ z.x.s do
     begin
          write "'n'Convergence attained in ",
          "iteration ",iformat( n ),".",
          "'nFinal triple ",
          "containing the zero of f is'n",
          tformat( [ z.x.i,z.m,z.x.s ],0,16 )
          abort
     end
     if z.m ≥ z.x.i and z.m ≤ z.x.s then
     x := [ z.x.i,z.m,z.x.s ]
     else
     x := [ z.x.i,( z.x.i + z.x.s ) / 2,z.x.s ]
     n := n + 1
end
write "'n'nMaximum number of iterations."
```

The above program produced the following results.

n	inf(x)	main(x)
0	-0.7000000000000001e+01	0.4711000000000000e+04
1	-0.7000000000000001e+01	0.2351500106112054e+04
2	-0.7000000000000001e+01	0.1171750265595868e+04
3	-0.7000000000000001e+01	0.5818755591461645e+03
4	-0.7000000000000001e+01	0.2869386373891154e+03
5	-0.7000000000000001e+01	0.1394710551756515e+03
6	-0.7000000000000001e+01	0.6573908703998034e+02
7	-0.7000000000000001e+01	0.2887703538153902e+02
8	-0.7000000000000001e+01	0.1045525295227102e+02
9	-0.7000000000000002e+01	0.1271274573832112e+01
10	-0.7000000000000002e+01	-0.1616139042072018e+01
11	-0.9983817225711355e+00	-0.1434128631756218e+00
12	-0.1798759659490610e-01	0.2056724932422962e-01
13	-0.1656910912515204e-01	-0.4230117447650240e-03
14	-0.1788632749431861e-06	0.1789389362091416e-06
15	-0.4960106059565740e-11	-0.3201914288863417e-13
16	-0.1028674620408200e-26	0.1025519176787315e-26

n	sup(x)
0	0.2479210000000001e+06
1	0.4710000212224109e+04
2	0.2350500531191738e+04
3	0.1170751118292330e+04
4	0.5808772747782310e+03
5	0.2859421103513031e+03
6	0.1384781740799607e+03
7	0.6475407076307805e+02
8	0.2791050590454205e+02
9	0.9542549147664226e+01
10	0.7115559962198918e+00
11	0.7115559962198919e+00
12	0.3574766401020007e+00

13 0.4144868895657036e-03
14 0.1394914761989317e-04
15 0.3201913718108796e-13
16 0.3166203683631262e-24

Maximum number of iterations.

Conclusions

The Triplex system was designed to be a simple but powerful extension to S-algol to provide high level facilities for interval arithmetic while retaining the power of the high speed floating point unit of most computers. The extensions to S-algol include facilities to manipulate triples as a basic data type, infix operators to operate on triples and standard functions for more complex triple operations. The system also protects triples from misuse and guarantees that they are well formed. This type of language extension leads to a concise notation that allows programs to be relatively small with all the attendant benefits that that entails.

To give the reader a feel for the utility of the system a solution to a well known problem in numerical analysis is outlined using the Triplex language. The system is available on request to the authors.

References

1. Alefeld, G. Uber die Existenz einer eindeutigen Losung bei einer Klasse nightlinearer Gleichungssysteme und deren berechnung mit Iterationsverfahren. Aplikace Matematiky 17,267-294 (1972).

2. Alefeld, G. & Herzberger, J. Nullstellereinschliessung mit dem Newton-verfahren ohne invertierung von Intervallmatrizen. Numerische Mathematik 19,56-64 (1972).

3. Apostolatos, N., Kulisch, U., Krawczyk, R., Lortz, B., Nickel, K. & Wipperman, H.W. The algorithmic language triplex algol 60. Numerische Mathematik 11,175-180 (1968).

4. Cole, A.J. & Morrison, R. An Introduction to Programming with S-algol. Cambridge University Press (1982).

5. Cole, A.J. & Morrison, R. Triplex: a system for interval arithmetic. Software, Practice & Experience 12,341-350 (1982).

6. Good, D.I. & London, R.L. Computer interval arithmetic. JACM 17,603-612 (1970).

7. Guenther, G. & Marquardt, G. A programming system for interval arithmetic in Algol 68, in Interval Mathematics 1980 (Nickel, K.L.E. (Ed)) (1980).

8. Gunn, H.I.E. & Morrison, R. On the implementation of constants. Information Processing Letters 9, (1979).

9. Hansen, E. (Ed) Topics in interval analysis. Oxford University Press (1969).

10. Hansen, E. A globally convergent interval method for computing and bounding real roots. BIT 18,415-424 (1978).

11. Hansen, E. Global optimisation using interval analysis - the one dimensional case. Journal of Optimisation Theory and Applications 29,331-344 (1979).

12. Krawczyk, R. Newton-algolithmen zur bestimmung von nullstellen mit fehlerschranken. Computing 4,1871201 (1969).

13. Landin, P.J. The next 700 programming languages. CACM 9,157-164 (1966).

14. Madsen, K. On the solution of non-linear equations in interval arithmetic. BIT 13,428-433 (1973).

15. Moore, R.E. Methods and applications of interval analysis. SIAM (1979).

16. Moore, R.E. Interval analysis. Prentice-Hall, New Jersey (1966).

17. Morrison, R. S-algol reference manual. University of St Andrews CS/79/1.

18. Nickel, K.L.E. (Ed) Interval Mathematics 1980. Academic Press (1980).

19. Nickel, K. Triplex algol and applications. In Topics in interval analysis (Hansen, E.R. ed.), Oxford University Press (1969).

20. Rall, L.B. A Comparison of the existence theorems of Kantorovich and Moore. SIAM Journal of Numerical analysis 17,148-161 (1980).

21. Strachey, C. Fundamental concepts in programming languages. Oxford University (1967).

22. Tennent, R.D. Language design methods based on semantic principles. Acta Informatica 8,97-112 (1977).

23. Wirth, N. The programming language Pascal. Acta Informatica 1,35-63 (1971).

24. Yohe, J.M. The INTERVAL arithmetic package. University of Wisconsin-Madison, Mathematics Research Centre MRC Technical Report No.1755 (1977).