Interval Computations
No 2, 1993

The VPI Software Package for Variable Precision Interval Arithmetic

Jeffrey S. Ely

The VPI (Variable Precision Interval) software package is a collection of routines written in C++ (by this author) to support variable precision interval arithmetic. It appears to be the oldest of the various C++ packages, having been used as early as 1988, although it has endured many modifications and enhancements since then. Here, the author discusses its capabilities, flaws, evolution (including the development of a vectorized version for the CRAY-YMP), and a variety of pedagogical and research applications to which the author has put it.

Пакет программ VPI для интервальной арифметики с переменной точностью

Дж. С. Или

Программный пакет VPI (Variable Precision Interval) является набором программ, написанных автором на C++ для поддержания интервальной арифметики с переменной точностью. Он, видимо, является самым старым из разнообразных пакетов на C++. Его использование началось в 1988 году, но с тех пор в него внесены многочисленные изменения и улучшения. Автор описывает его возможности, недостатки, эволюцию (в том числе разработку векторизованной версии для CRAY-YMP), а также множество педагогических и исследовательских приложений, для которых автор его предназначал.

1 Introduction

The VPI software package is a collection of routines written in C++ to support variable precision interval arithmetic. This author has applied it to a problem in fluid dynamics [1]. That application shaped much of its current design and also motivated the development of a vectorized version for a CRAY-YMP.

2 Capabilities of VPI

The software package VPI (Variable Precision Intervals) is structured as shown in Figure 1. At the bottom are low level procedures for manipulating bit strings. On top of this, variable precision floating-point numbers (`afloat`) are constructed, while on top of this rests intervals (`interval`). On top of the interval layer, there are three extensions: complex intervals (`compivl`), matrices (`matrix`), and Taylor series (`taylor`). Additionally, complex Taylor series (`ctaylor`) are built from the complex intervals.

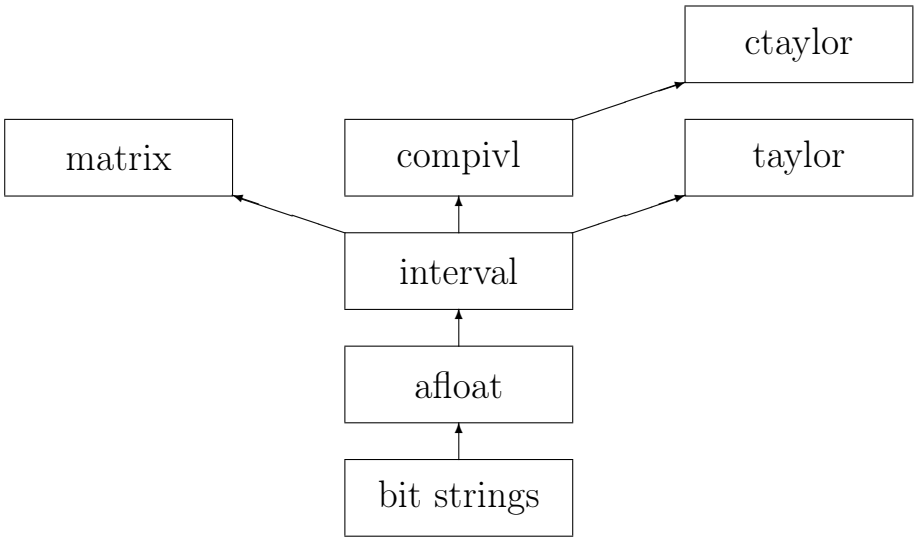

Figure 1: The data types of VPI and their dependence.

Bit strings. The lowest level contains routines to manipulate arbitrarily long strings of bits. Bit strings can be moved, cleared, added, subtracted, multiplied, divided, rotated left or right, or tested for zero. The routines for doing these operations are `bimov`, `biclear`, `biadd`, `bisub`, `biunsmul`, `biunsdiv`, `bircl`, `bircr`, and `bizerotest`, respectively. Typically, these routines require the starting address of the bit strings involved and the length (in words) of the strings. This layer will seldom interest high level users, but is the one where most of the effort at efficiency is spent. In early versions of VPI, these routines were written in assembly language, although the current version is all C++ code for portability. This is also the layer that was vectorized on a CRAY-YMP (more on this later).

Variable precision floating-point numbers. The original plans for VPI called for truly arbitrary precision floating-point numbers, limited only by available memory, hence the name of the data type `afloat`. Early timing tests indicated an inordinate amount of time was being consumed in the memory management phase of these procedures, so arbitrary precision was dropped in favor of variable precision up to a limit (currently 34 words). A variable of type `afloat` is an array of 34 words, not all of which are used. A word is reserved for the exponent, one for the sign, and one to indicate the number of other words being used for the mantissa. This mantissa length was incorporated for an earlier design where some variables might have 3 words of mantissa, others might have 7 or 8. This scheme was abandoned in favor of a global variable `lenMANT`, which indicates the number of words currently being used by *all* variables of type `afloat`.

Reserving an entire word (32 bits) for the exponent means that numbers can become quite large and further implies the infeasibility of an accurate dot product such as exists in Pascal–SC [5], Pascal–XSC [3], or C–XSC [4]. The sign word has 3 possible values; -1 indicating a negative number, $+1$ for a positive number, 0 if the number is zero. Original plans to support -2 for $-$infinity, $+2$ for $+$infinity were never realized. The binary point is assumed to be immediately prior to the bits of the mantissa. Hence, a normalized mantissa represents a number in $[\frac{1}{2}, 1)$.

There are five categories of procedures that operate on `afloat`s: constructors, I/O routines, arithmetic operators and functions, relational operators, and a few oddities. A C++ constructor exists for constructing an `afloat` from an int. The procedure `dblfromafl` will construct a double

from an `afloat` (if possible). This is useful for producing results that conventional programs can then read. For input and output, the usual C++ operators `>>` and `<<` have been overloaded to give conversions between the internal binary representation and an external decimal one. No effort is made for directed rounding of these results. If directed rounding *is* desired, the procedures `dirin` and `dirout` permit the specification of rounding direction. For exact I/O, `aflhexin` and `aflhexout` read and write the data in a hexadecimal form. This is seldom pleasant to read, but very handy as a medium of exchange between two different programs written in VPI, enabling the sharing of data via files without the round-off errors that a decimal exchange would incur.

The full set of relational and arithmetic operators, `==`, `!=`, `<`, `<=`, `>`, `>=` , `+`, $-$, $*$, and `/`, have been overloaded to operate on `afloat`s. The arithmetic operators do not permit a rounding direction to be specified, but the alternate procedures, `dirplus`, `dirminus`, `dirmul`, and `dirdiv`, do. While not as convenient to use for complicated expressions as their operator counterparts, these procedures are (appropriately) more thought-provoking and form the basis for the subsequent interval arithmetic. The absolute value, `abs`, change sign, `chs`, and `mulbypow2` (multiplication by a power of two) are all exact.

A few final oddities should be mentioned. The `separate` procedure extracts the integer part from the fractional part of an `afloat`, whenever the integer part will fit into one word. This turns out to be handy in doing decimal I/O and later in the interval calculation of trigonometric and exponential values.

For something as simple as this array definition of an `afloat`, it is not really necessary for VPI to have its own code for the assignment operator `=` since C++ will default to a bitwise copy of the entire array. However, this default is not satisfactory for two reasons. First, if the precision being used is low (say `lenMANT` = 3, implying only 3 words of mantissa), copying all 34 words is inefficient. The second reason is specific to the CRAY. It is unclear if the default C++ data movement procedure has been vectorized on the CRAY, whereas VPI's `bimov` definitely is.

The interval data type. An `interval` is defined as two `afloat`s, a left and a right. An `interval` may be constructed from an int, from a single `afloat` (which gives an `interval` with left part equal to the right part), or

from a pair of `afloat`s (one for the left part, one for the right).

The I/O operators, `>>` for input and `<<` for output, bound the errors as they convert between the internal binary format and an external decimal format. Procedures `ivlhexin` and `ivlhexout` are analogous to their `afloat` counterparts and provide exact hexadecimal I/O.

The arithmetic operators `+`, $-$, $*$, and `/` have been overloaded with their obvious definitions. Additionally, there is a reasonable selection of mathematical functions, `abs`, `arccos`, `arccosh`, `arctan`, `cos`, `cuberoot`, `exp`, `ln`, `mulbypow2`, `sin`, `sqrt`, and `square`. The square function does the correct thing for intervals, as opposed to merely implementing x$*$x. The function `mulbypow2` multiplies by a power of 2 and is exact, as is the absolute value function `abs`. The functions `sqrt` and `cuberoot` are implemented with an interval version of Newton's method, while the transcendental functions are built on power series (since the error terms are nicely computed). The non-monotonic nature of `sin` and `cos` required special care to ensure both correctness and "tightness".

While the procedures `leftpart`, `rightpart`, `mid`, and `width` should be obvious, the procedure `intersect`, for intersecting two intervals, requires some explanation. A program might invoke it as

```
signal = intersect(iV1, iV2, iV3)
```

where iV2 and iV3 are the two intervals being compared, and iV1 is to be their intersection. The returned int, signal, is

0 if iV2 and iV3 are equal,
1 if they do not overlap, in which case iV1 is left alone,
2 if iV2 is properly contained in iV3,
3 if iV3 is properly contained in iV2, and
4 otherwise.

This is cumbersome but thought-provoking to use.

The user may select the precision to be used by VPI in two ways. If no transcendental functions are to be used, a simple change to the global constant `lenMANT` will establish the number of words to be used for mantissas. For instance,

```
lenMANT = 5 ;
```

will cause 5 words of mantissa to be used in all subsequent calculations. On most machines, this will mean $5 * 32 = 160$ bits of precision. However, on the CRAY, this will mean $5 * 64 = 320$ bits of precision. If transcendental functions are to be invoked, the user should establish the precision by a call to the function `precision` such as

```
precision(5) ;
```

This call changes the constant `lenMANT` and also alters certain other global constants such as `pi` and `lntwo` (the natural log of 2). Normally, the precision is established at the beginning of a program, before even any variable declarations of type afloat or interval are made and is the same throughout the running of the program. The user can, however change the precision in midstream (although with care). If no transcendental functions are involved, the simple reassignment of `lenMANT` is very fast, but if transcendental functions are involved, `precision` must be called. This can be quite slow, especially if the new precision is large.

Complex intervals. Several different data types have been built on top of type `interval`. One of these is `compivl`, the complex interval, with an `interval` real part and an `interval` imaginary part.

Complex numbers can be constructed directly from ints, from a single `interval` (which assigns this single interval to the real part and sets the imaginary part to zero), or from a pair of `interval`s (the first specifies the real part, the second, the imaginary part).

The I/O operators `>>` and `<<` allow for decimal input and output, but there are no hexadecimal I/O functions, as there are for `afloat`s and `interval`s. The functions `realpart` and `imaginarypart` return the real and imaginary components of `compivl`s. For arithmetic, VPI supports `+`, $-$, $*$, and `/`, `arccos`, `arccosh`, `arctan`, `cos`, `exp`, `ln`, and `sqrt`. Additionally, `conjugate` and `modulus` are available. `Modulus` carefully computes $\mathrm{sqrt}\big(\mathrm{square}(a) + \mathrm{square}(b)\big)$ instead of $\mathrm{sqrt}(a * a + b * b)$ to avoid possibly taking the square root of an interval containing negative numbers.

The only oddity is the function `cossin`. This is to be used when the cosine and sine of the same argument are desired. For example, the call `cossin (cs,sn,x)` is equivalent to, but more efficient than, `cs = cos(x); sn = sin(x)`.

A favorite way to test VPI is to check the results of both sides of mathematical identities. If the interval results of the two sides do not overlap, then there is a bug somewhere. Continued overlap of the two sides at higher and higher precisions builds confidence in the correctness of the code. The

file civlTEST.c (below) shows one such test, computing both sides of the complex number identity

$$\sum_{k=0}^{n-1} \cos(kt) + i\sin(kt) = \frac{1 - e^{int}}{1 - e^{it}}.$$

civlTEST.c

```
100   #include <compivls.h>

110   main()
120   {
130    int k ,n = 10 ;
140    precision(3) ; // select 3 word mantissas

150    compivl s, check, t, i = compivl(0,1);

160    cout << "\n enter t(complex interval)" ;
170    cin >> t ;
180    cout << "\nt = " << t ;

190    s = 0 ;
200    for ( k = 0 ; k < n ; k++ ) {
210         s = s + cos(k*t) +i*sin(k*t) ;
220    }

230    check = (1 - exp(i*n*t)) / (1 - exp(i*t)) ;
240    cout << "\ns     = " << s ;
250    cout << "\ncheck = " << check ;
260   }
```

```
civlTEST.res

precision of 96 bits.
enter t(complex interval)

t =      [ +0.99999999999999999999999999959e+0 ,
           +0.10000000000000000000000000005e+1 ]
   + i   [ +0.19999999999999999999999999995e+1 ,
           +0.20000000000000000000000000010e+1 ]

s    =   [ +0.10628461257270163104292004453 6e+1 ,
           +0.10628461257270163113444195448 9e+1 ]
   + i   [ +0.13058641198434409485821883830 2e+0 ,
           +0.13058641198434409563600684211 8e+0 ]

check =  [ +0.10628461257270163108869234992 5e+1 ,
           +0.10628461257270163108869235004 3e+1 ]
   + i   [ +0.13058641198434409524718213115 4e+0 ,
           +0.13058641198434409524718213161 9e+0 ]
```

Line 100 shows the inclusion of the proper include file compivls.h that contains the definitions of complex interval (`compivl`) operators and functions. Line 140 sets the precision to 3 words of 32 bits. Notice that this comes before the declarations of the `compivls`, `s`, `check`, `t`, and `i`. Line 150 further initializes `i` via a call to the constructor

```
i = compivl (0, 1) ;
```

The precision needs to be established *before* this initialization.

Line 170 waits for the user to enter a complex value for `t`. A typical response would be [1, 1] + i [2, 2]. Line 180 immediately echoes the input. Notice in the output file civlTEST.res that the I/O combination has introduced some small error in merely echoing `t`.

Lines 190 through 220 compute the left hand side of our identity,

$$\sum_{k=0}^{n-1} \cos(kt) + i \sin(kt)$$

while line 230 computes the right hand side, $\frac{1-e^{int}}{1-e^{it}}$. This code is entirely as natural as if complex intervals were a built-in type.

Matrices. Another layer built on top of the type interval is type `matrix`. Beyond the usual I/O and arithmetic operators, `>>`, `<<`, `+`, $-$, $*$, `/`, and the functions,

`int numrows(matrix & )`,
`int numcols(matrix & )`,
`interval norm(matrix & )` (Euclidian norm), and
`matrix transpose(matrix & )`,

the constructor,

`matrix(int, int)`,

permits the user to construct a matrix with specified numbers of rows and columns at run time (instead of at compile time). This is the only data type in VPI that performs run time memory management.

The program conditionnumTEST.c computes the condition number of an $m \times m$ Hilbert matrix. It indicates how to use many of the matrix class procedures, including the obscure `dptrptr`, which is needed to manipulate the individual components of a matrix. The output of a test run can be found in conditionnumTEST.res.

`conditionnumTEST.c`

```
100    #include <matrix.h>

110    void makehilbert(matrix& u)
120    // makes Hilbert matrix in left half,
       // identity in right
130    {
140      int i,j ;
150      int m = numrows(u) ;
160      int n = numcols(u) ;      // should be = 2*m
170      DPTRPTR up = dptrptr(u) ;

180      for (i=0;i<m;i++) {
190         for (j=0;j<m;j++) {
200             up[i][j] = interval(1)/interval(i+j+1) ;
210         }
220         for (j=m;j<n;j++) {
230             if (i==j-m) { up[i][j] = interval(1) ; }
```

```
240             else { up[i][j] = interval(0) ; }
250          }
260       }
270    } // end makehilbert


280    main()
290    {
300       lenMANT = 1 ;        // 1 word of precision
310       int m = 3 ;          // numrows of Hilbert matrix

320       matrix mat(m,2*m) ;  // create the space
330       makehilbert(mat) ;

340       matrix H(m,m) ;
350       embed(H,0,0,mat,0,0,m-1,m-1) ;

360       solve(mat) ;

370       if (errval(mat) == 0) {
380         cout << "\nCan't invert with given precision." ;
390         exit(0) ;
400       }

410       matrix Hi(m,m) ;
420       embed(Hi,0,0,mat,0,m,m-1,n-1) ;

430       matrix Icheck(m,m) ;
440       Icheck = H*Hi ;

450       interval condition = norm(H)*norm(Hi) ;

460       cout << "\n\nHilbert matrix, H    = " << H ;
470       cout << "\n\nInverse matrix, Hi   = " << Hi ;
480       cout << "\n\nIdentity check, H*Hi = " << Icheck ;
490       cout << "\n\n\ncondition number   = " << condition ;
500    }
```

`conditionnumTEST.res`

```
Hilbert matrix, H    =
[ +0.099999999962e+1,  [ +0.500000000000e+0,  [ +0.333333333116e+0,
  +0.100000000046e+1 ]   +0.500000000000e+0 ]   +0.333333333488e+0 ]

[ +0.500000000000e+0,  [ +0.333333333116e+0,  [ +0.250000000000e+0,
  +0.500000000000e+0 ]   +0.333333333488e+0 ]   +0.250000000000e+0 ]

[ +0.333333333116e+0,  [ +0.250000000000e+0,  [ +0.199999999925e+0,
  +0.333333333488e+0 ]   +0.250000000000e+0 ]   +0.200000000186e+0 ]


Inverse matrix, Hi   =
[ +0.899999878555e+1,  [ -0.360000074561e+2,  [ +0.299999934136e+2,
  +0.900000134110e+1 ]   -0.359999931976e+2 ]   +0.300000071991e+2 ]

[ -0.360000018961e+2,  [ +0.191999993100e+3,  [ -0.180000010207e+3,
  -0.359999987855e+2 ]   +0.192000010535e+3 ]   -0.179999993275e+3 ]

[ +0.299999988339e+2,  [ -0.180000010207e+3,  [ +0.179999993499e+3,
  +0.300000018440e+2 ]   -0.179999993313e+3 ]   +0.180000009872e+3 ]


Identity check, H*Hi =
[ +0.999997459352e+0,  [ -0.142157077789e-4,  [ -0.137686729431e-4,
  +0.100000254111e+1 ]   +0.142157077789e-4 ]   +0.137686729431e-4 ]

[ -0.152364373207e-5,  [ +0.999991461634e+0,  [ -0.827014446258e-5,
  +0.152550637722e-5 ]   +0.100000853883e+1 ]   +0.827014446258e-5 ]

[ -0.110827386379e-5,  [ -0.621378421783e-5,  [ +0.999993979930e+0,
  +0.111013650894e-5 ]   +0.619888305664e-5 ]   +0.100000603543e+1 ]


condition number = [ +0.276843083277e+6,
                     +0.276843137852e+6 ]
```

After the user specifies the precision of the interval arithmetic (line 300) and the matrix size `m` (line 310), line 320 invokes a constructor to create a matrix called `mat` of size $\texttt{m} \times \texttt{n}$, where $\texttt{n} = 2 * \texttt{m}$. Line 330 calls a procedure to initialize the left half of `mat` with a Hilbert matrix and the right half with the identity matrix. The code for this procedure, `makehilbert`, is also given (see lines 110 to 270), mainly to show how to use the function `dptrptr` to

obtain the address where the data of a matrix is stored. Once obtained (in line 170), lines 200, 230, and 240 show how to naturally use this address and common matrix subscripting to access arbitrary elements of the matrix.

Returning to the main program, lines 340 and 350 extract the left half (the Hilbert half) of matrix `mat` and save it in matrix `H` using procedure `embed`. Line 360 calls the procedure `solve` to perform Gaussian elimination on `mat`. If successful, the right half of `mat` should now contain the inverse, which is extracted by `embed` and saved in `Hi` in lines 410 and 420. If, for some reason, `solve` cannot do its job, it sets an error code in the err field of the matrix `mat` to 0. This can be checked by the function `errval`, as in lines 370 through 400. If all goes well, lines 430 and 440 compute `H * Hi` (which ideally would be the identity matrix). The condition number $= norm(\texttt{H}) * norm(\texttt{Hi})$ is computed on line 450, and various information is output in subsequent lines.

The file conditionnumTEST.res shows the details of `H`, `Hi`, and `H * Hi`, and the condition number for the 3×3 Hilbert matrix using 32 bits of precision. In order to explore Hilbert matrices of size 8 or more, the high precision that VPI affords becomes critical.

Taylor series arithmetic. A third extension of class `interval` is class `taylor`. This is a realization of recursive differentiation arithmetic which is sometimes called automatic differentiation [2].

The file tay.example.c shows how to manipulate this class in VPI.

`tay.example.c`

```
100  #include <taylor.h>

110  main()
120  {
130   precision(1) ; // use 1 word (32 bits) of precision
140   int n = 6 ; // degree of taylor series

150   taylor c = taylor(interval(1),0,n) ;
      // deg n, 0th term = 1
160   taylor x = taylor(interval(1),1,n) ;
      // deg n, 1st term = 1
```

```
170   taylor expc = exp(c) ;
180   taylor expx = exp(x) ;
190   taylor s = sin(x) ;
200   taylor y = 1/(1+x*x) ;
210   taylor z = 2*cos(x)*s ;

220   interval expsum = taysum(expx) ;
230   taylor txivl =
          taylor(interval(0,1),0,interval(1),1,n+1) ;
240   taylor terr = exp(txivl) ;
250   interval errorterm = termselect(terr,n+1) ;

260   cout << "\ndegree "  << n << " selected." ;
270   cout << "\nexp(x)          = " << expx ;
280   cout << "\nsin(x)          = " << s ;
290   cout << "\n1/(1+x*x)      = " << y ;
300   cout << "\n2*cos(x)*sin(x) = " << z ;
310   cout << "\nsum of terms of exp(x)   = " << expsum ;
320   cout << "\nthe error term of exp(x) = " << errorterm ;
330   cout << "\ninclusion of e  = " << expsum + errorterm ;
340  }
```

After declaring the precision of the interval arithmetic to be used in subsequent calculations in line 130, the degree `n` of the Taylor series is set to 6 in line 140. Lines 150 and 160 invoke the class constructor

`taylor (interval & , int, int)` ,

to initialize

`c`$= [1, 0, 0, 0, 0, 0, 0]$, and
`x`$= [0, 1, 0, 0, 0, 0, 0]$.

Lines 170, 180 compute

$e^{\texttt{c}} = [e^1, 0, 0, 0, 0, 0, 0]$, and
$e^{\texttt{x}} = [1, \frac{1^1}{1!}, \frac{1^2}{2!}, \frac{1^3}{3!}, \frac{1^4}{4!}, \frac{1^5}{5!}, \frac{1^6}{6!}]$.

The output file tay.example.res shows some of these calculations in interval form. By comparing the program and the output, the reader will also see

$\sin(\texttt{x}) = \left[0, 1, 0, \frac{-1^3}{3!}, 0, \frac{+1^5}{5!}, 0\right]$
$\frac{1}{1+\texttt{x}^2} = \left[1, 0, -1^2, 0, +1^4, 0, -1^6\right]$

and

$2 * \cos(\mathtt{x}) * \sin(\mathtt{x}) = \left[0, 2, 0, \frac{-2^3}{3!}, 0, \frac{2^5}{5!}, 0\right]$ (recall $= \sin 2\mathtt{x}$).

Line 220 invokes the function,

`interval taysum(taylor & )`,

to sum the terms of $e^{\mathtt{x}}$, producing an approximation to e^1. Line 230 invokes another constructor to initialize the variable

$\mathtt{txivl} = \left[[0, 1], 1, 0, 0, 0, 0, 0, 0\right]$

of degree $n + 1$, which is 7 in this example, one more than for `x` or $e^{\mathtt{x}}$. Line 240 computes $e^{\mathtt{txivl}}$, and line 250 extracts the term of degree 7. This term added to the earlier sum of the 7 terms of $e^{\mathtt{x}}$ (see line 330), gives a guaranteed inclusion for e^1 (see output file tay.example.res).

`tay.example.res`

```
precision of 32 bits.     degree 6 selected.

exp(x)          =
 [  [ +0.099999999962e+1 , +0.100000000046e+1 ]  ,
    [ +0.099999999962e+1 , +0.100000000046e+1 ]  ,
    [ +0.500000000000e+0 , +0.500000000000e+0 ]  ,
    [ +0.166666666548e+0 , +0.166666666753e+0 ]  ,
    [ +0.416666666232e-1 , +0.416666666977e-1 ]  ,
    [ +0.833333332091e-2 , +0.833333333954e-2 ]  ,
    [ +0.138888888619e-2 , +0.138888888992e-2 ]  ]

sin(x)          =
 [  [ -0.351150604244e-7 , +0.415730120986e-7 ]  ,
    [ +0.999999930337e+0 , +0.100000000046e+1 ]  ,
    [ -0.207865060493e-7 , +0.175575302131e-7 ]  ,
    [ -0.166666666753e+0 , -0.166666655093e+0 ]  ,
    [ -0.146312751807e-8 , +0.173220883831e-8 ]  ,
    [ +0.833333274349e-2 , +0.833333333954e-2 ]  ,
    [ -0.577402946352e-10 , +0.487709173001e-10 ]  ]

1/(1+x*x)       =
 [  [ +0.099999999962e+1 , +0.100000000046e+1 ]  ,
```

```
    [ +0.000000000000e+0 , +0.000000000000e+0 ]  ,
    [ -0.100000000046e+1 , -0.099999999962e+1 ]  ,
    [ +0.000000000000e+0 , +0.000000000000e+0 ]  ,
    [ +0.099999999962e+1 , +0.100000000046e+1 ]  ,
    [ +0.000000000000e+0 , +0.000000000000e+0 ]  ,
    [ -0.100000000046e+1 , -0.099999999962e+1 ]  ]

2*cos(x)*sin(x) =
 [  [ -0.702301208674e-7 , +0.831460242718e-7 ]  ,
    [ +0.199999972246e+1 , +0.200000000186e+1 ]  ,
    [ -0.166292048487e-6 , +0.140460241772e-6 ]  ,
    [ -0.133333333441e+1 , -0.133333314768e+1 ]  ,
    [ -0.468200805969e-7 , +0.554306828416e-7 ]  ,
    [ +0.266666629351e+0 , +0.266666667070e+0 ]  ,
    [ -0.739075771532e-8 , +0.624267741665e-8 ]  ]


sum of terms of exp(x)   =  [ +0.271805555280e+1 ,
                              +0.271805555839e+1 ]

the error term of exp(x) =  [ +0.198412698060e-3 ,
                              +0.539341636002e-3 ]

inclusion of e           =  [ +0.271825396548e+1 ,
                              +0.271859490033e+1 ]
```

Complex Taylor series. The last class currently supported by VPI is `ctaylor`, which, as its name suggests, is the complex version of `taylor` and requires no special discussion. Although `ctaylor` is the last class which VPI supports, it should be clear that users of VPI can implement their own classes with ease. Beyond the examples given here, serious users should consult Stroustrup [7] for details about C++. This is a small book, well written, offering numerous examples. Anyone familiar with Pascal, Fortran, or "C", will quickly comprehend C++.

3 A vectorized version for the CRAY-YMP

Originally, VPI was written for an HP-9000 Unix workstation, and the bit string layer was written in assembly language for speed. Shortly thereafter, an opportunity arose to apply VPI to a problem in fluid dynamics [6]. It was felt that a vectorized version for a CRAY-YMP might give rise to a much needed speedup. Again, the low level bit string routines were the logical candidates for vectorization, so these routines were re-written in C++, and much time was spent trying to vectorize them. Vectorization is most easily implemented when some simple operation must be performed on every element in an array. While these simple operations are not done on each array element in true parallel fashion, yet it is true that the operation on x[7] may have commenced before that same operation on x[6] is finished. For such operations as array movement (`bimov`), no complications arise, but for an operation such as array addition (`biadd`), where the carry out of adding x[6] and y[6] will affect the result of adding x[7] and y[7], special care must be taken. Trickier still are the multiplication and (especially) the division operations.

The vectorization effort produced substantial improvement over either the HP or the non-vectorized CRAY version (it is possible to compile with the vectorization turned off), but is disappointing when compared to a workstation of more recent vintage such as a DEC 5000.

Table 1 compares running times of the vectorized CRAY-YMP version versus the DEC 5000 version on various mathematical functions. Table 2 compares the two on the problem of inverting a 15×15 Hilbert matrix using various precisions, while Table 3 compares the two on the program civlTEST.c, which makes heavy use of the complex interval transcendental functions (see the earlier section on complex interval arithmetic).

As might be expected at low precisions, very little is gained by vectorization. For example, inverting the 15×15 Hilbert matrix at 192 bits of precision (6 words of 32 bits for a DEC 5000 but only 3 words of 64 bits for a CRAY-YMP) took 1.38 seconds on the CRAY and only a little longer (2.30 seconds) on the DEC. The higher precision of 768 bits gives 8.94 seconds for the CRAY and 27.20 seconds for the DEC, i.e. the CRAY is only 1.67 times as fast as the DEC for 192 bits but is 3.04 times as fast for 768 bits. The 768 bits is still only 12 CRAY words and is considered a "small" vector. One expects that for still larger vectors (higher precisions), the improvement

	CRAY-YMP			DEC 5000		
	64 bits	256 bits	768 bits	64 bits	256 bits	768 bits
+	70	100	100	50	100	200
−	80	100	100	80	100	200
∗	90	200	400	80	400	3,200
/	340	1,800	12,400	430	3,500	29,200
sqrt	10,000	50,000	340,000	11,000	90,000	740,000
exp	30,000	210,000	2,800,000	20,000	300,000	5,200,000
ln	30,000	310,000	5,200,000	20,000	530,000	11,100,000
cos	50,000	490,000	7,100,000	60,000	840,000	13,700,000
sin	60,000	500,000	7,000,000	60,000	830,000	13,800,000

Table 1: Timing of various interval math functions (in microseconds).

	CRAY-YMP	DEC 5000
192 bits	1.38	2.30
384 bits	2.97	7.30
768 bits	8.94	27.20

Table 2: Timing comparison: inversion of 15 × 15 Hilbert matrix (in seconds).

would be more marked, but most of the author's applications so far have not required precisions much larger than the 768 bits. On these problems, it makes little sense to consume expensive CRAY time when a workstation is available.

4 Applications

To date, the author's only research application of VPI has been the previously mentioned problem in fluid dynamics. Several pedagogical opportunities have surfaced as illustrated in the examples of the condition number and the calculation of an inclusion of e by Taylor's theorem. Additionally, VPI is ideal for use by students to numerically calculate limits or to find

$$\sum_{k=0}^{n-1} \cos(kt) + i\sin(kt) = \frac{1 - e^{int}}{1 - e^{it}}, \qquad \text{with } n = 10,\ t = [1,1] + i[2,2]$$

	CRAY-YMP		DEC 5000	
64 bits	4.78		5.10	
256 bits	42.08		70.50	
768 bits	600.19	(10 minutes)	1199.90	(20 minutes)

Table 3: Timing comparison: complex transcendental identity (in seconds).

formulae via the method of undetermined coefficients. Both of these applications frequently challenge the numerical soundness of typical calculators and programming languages.

5 Future directions

Several future projects involving VPI have occurred to this author. One would be to write and freely distribute over the internet some sort of software interval calculator such as xcalc. This might help familiarize a larger audience with at least naive interval computation.

Another project would extend VPI's interval data type to also support hardware-based interval computation. Perhaps a specification of 0 precision could invoke the underlying floating-point hardware instead of the software intensive `afloat`s. The virtue of this would be speed when only low precision was required.

A final suggestion would be to explore parallelism via a network of workstations. The use of a CRAY has lost its appeal, but a network approach to parallelism might prove practical.

References

[1] Ely, J. and Baker, G. *High-precision calculations of vortex sheet motion.* Submitted 1992.

[2] Kagiwada, H., Kalaba, R., Rasakhoo, N., and Spingarn, K. *Numerical derivatives and nonlinear analysis.* Plenum, New York, 1986.

[3] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch. *Pascal–XSC: language reference with examples.* Springer-Verlag, Berlin, 1992.

[4] Klatte, R., Kulisch, U., Lawo, Ch., Rauch, M., and Wiethoff, A. *A C++ class library for extended scientific computation.* Springer-Verlag, Berlin, 1993.

[5] Kulisch, U. *Pascal–SC, a Pascal extension for scientific computation.* Wiley, New York, 1987.

[6] Meiron, D., Baker, G., and Orszag, S. *Analytic structure of vortex sheet dynamics. Part 1. Kelvin-Helmholtz instability.* J. Fluid Mechanics **114** (1982), pp. 283–298.

[7] Stroustrup, B. *The C++ programming language.* Addison-Wesley, Reading, Mass., 1986.

Department of Mathematical Sciences
Lewis and Clark College
Portland, OR 97219
USA
E-mail: `jeff@lclark.edu`