Bergische Universität
Wuppertal

Advanced Software Tools for Validated Computing

Walter Krämer

Preprint 2002/1

Wissenschaftliches Rechnen/
Softwaretechnologie

wr
swt

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich 7 (Mathematik) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Prof. Dr. Walter Krämer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `kraemer@math.uni-wuppertal.de`

Advanced Software Tools for Validated Computing

Walter Krämer
Scientific Computing/Software Engineering
University of Wuppertal, Germany

Abstract: Validated computing based on interval computations is one essential technology to achieve increased software reliability. In this paper several advanced interval software tools with emphasis on validated computing are considered. Pros and cons are given and by sample codes the usage of the tools is illustrated (for your convenience the full codes will be made available on our web pages `http://www.math.uni-wuppertal.de/wrswt/`).

The tools INTLAB, Sun's Forte C++ compilers with interval support, filib++, and C-XSC are discussed in some detail. Due to limited space, many other interval tools for more or less special applications are not considered here. You can find pointers to such tools on the web (e. g. `http://www.cs.utep.edu/interval-comp/intsoft.html`).

1 Introduction

The program committee organizing `Validated Computing 2002` wrote

> Ever increasing reliance on computer systems brings ever increasing need for reliability. Validated computing is one essential technology to achieve increased software reliability. Validated computing uses controlled rounding of computer arithmetic to guarantee that hypotheses of suitable mathematical theorems are (or are not) satisfied. Mathematical rigor in the computer arithmetic, in algorithm design, and in program execution allow us to guarantee that the stated problem has (or does not have) a solution in an enclosing interval we compute. If the enclosure is narrow, we are certain that we know the answer reliably and accurately. If the enclosing interval is wide, we have a clear warning that our uncertainty is large, and a closer study is demanded.
>
> Intervals capture uncertainty in modeling and problem formulation, in model parameter estimation, in algorithm truncation, in operation round off, and in model interpretation.

In this paper we discuss pros and cons for several advanced interval software tools with emphasis on validated computing. Our sample codes for simple but powerful applications illustrate the usage of the tools INTLAB [11, 12], Sun's Forte C++ compilers with interval support [14, 15, 16], filib++ [8, 9], and C-XSC [3, 5]. Other interval tools for more or less special applications (see e. g. `http://www.cs.utep.edu/interval-comp/intsoft.html`) are not considered here. Those who are interested in recent developments in the field of Validated Computing may consult [6, 7] and the literature cited therein.

2 The INTerval LABoratory INTLAB

INTLAB [11, 12] is a well designed interval toolbox for the interactive programming environment MATLAB [10]. It allows the more traditional infimum-supremum as well as the midpoint-radius representations of intervals. Operators for mixed operands are available:

```
intvalinit('DisplayInfsup'); % use inf-sup notation for output
midrad(2,1) + infsup(3,4)    % operands may also be vectors or matrices
```

produces the output `[4.000, 7.000]`.

The midpoint-radius interval arithmetic of INTLAB is entirely based on BLAS. Matrix and vector operations avoid in a clever way case distinctions and the time consuming switching of rounding mode in inner loops at the expense of some additional BLAS operations. So, in particular, INTLAB matrix operations are very fast. In contrast to traditional infimum-supremum arithmetic the midpoint-radius implementation takes full advantage of the speed of vector and parallel architectures. The (theoretical) overestimation of midpoint-radius arithmetic compared to infimum-supremum arithmetic is globally limited by a factor 1.5 for the basic arithmetic operations as well as for vector and matrix operations (independent of the dimension of the matrices) over $\mathbb{IR}$ and $\mathbb{IC}$ [12]. In practical machine computations the factor is around 1.0, and sometimes even less than 1 (due to finite precision machine arithmetic).

Every computation using INTLAB is rigorously verified to be correct, including input and output. Portability is assured by implementing all algorithms in MATLAB itself with exception of exactly three routines for switching the rounding downwards, upwards and to nearest (the routines for switching the rounding mode are freely available for many platforms on the web). INTLAB itself may be freely copied from the home page `http://www.ti3.tu-harburg.de/~rump/intlab`. But to be able to use INTLAB you have to buy the commercial product MATLAB [10] (see `http://www.mathworks.com`).

INTLAB allows to write verification algorithms in a way which is very near to pseudo-code used in scientific publications. E. g., the following INTLAB code may be used to solve a dense system of linear equations with automatic result verification (the code is taken from the file `verifylss.m` coming with INTLAB).

```
function X = denselss(A,b) % linear system solver for dense matrices
  midA = mid(A);           % midpoint matrix
  midb = mid(b);

  R = inv( midA ) ;        % preconditioner: approximate inverse

  xs = R * midb ;          % approximate solution

  % interval iteration
  A = intval(A);
  Z = R * (b - A*xs) ;
  RA = R*A;
```

```
  C = eye(dim(A)) - RA;    % eye produces an identity matrix
  Y = Z;
  E = 0.1*rad(Y)*hull(-1,1) + midrad(0,10*realmin); % prepare inflation
  k = 0; kmax = 7; ready = 0;
  while ( ~ready ) & ( k<kmax ) & ( ~any(isnan(Y(:))) )
    k = k+1;
    X = Y + E;                  % inflation
    Y = Z + C * X;
    ready = all(all(in0(Y,X))); % check proper inclusion
  end
  if ready
    X = xs + Y;                 % verified result
  else                          % no succss
    disp('*** In routine denselss: No verification!')
    X = NaN;
 end
```

Let this code be stored in a Matlab M-File `denselss.m`. Then we can call this routine to compute a verified enclosure of the solution of a linear system:

```
intvalinit('DisplayInfsup'); % use inf/sup output
===> Default display of intervals by infimum/supremum (e.g. [ 3.14 , 3.15 ])
A = random(3);               % generate a 3 by 3 point matrix
A = hull(1-1e-3, 1+1e-3)*A   % intervalmatrix
intval A =
[    0.2543,    0.2549] [   -0.1730,   -0.1725] [   -0.2571,   -0.2565]
[    0.3977,    0.3986] [    0.3101,    0.3108] [   -0.1497,   -0.1493]
[   -0.2059,   -0.2054] [    0.6744,    0.6759] [    0.1891,    0.1896]
b = ones(3,1)                % right hand side
b =
     1
     1
     1
solution = denselss(A,b)     % solve the linear system
intval solution =
[   -3.4063,   -3.3193]
[    3.0485,    3.0745]
[   -9.3545,   -9.2224]
all( in(b, A*solution) )     % check b  element of  A*solution
ans = 1
```

Note, the code for `denselss()` works for real point and real interval matrices as well as for complex point and complex interval matrices.

INTLAB offers predefined problem solving routines for dense and sparse systems of linear and nonlinear equations and eigenvalue problems (`verifylss`, `verifynlss`, `verifyeig`). A multi precision interval arithmetic, a slope arithmetic as well as routines for automatic differentiation are also included. The main features of INTLAB can

be nicely explored using several demo files (`demointval`[1], `demolong`[1], `demoslope`[1], `demogradient`[1]).

INTLAB is a very powerful interactive tool to implement prototypes of verification algorithms. INTLAB code is elegant, easy to read and to maintain. For those who are interested in guaranteed numerical results and have available MATLAB the INTLAB package is a must.

3 Sun's Forte C++ compilers with interval support

In this section features of the interval library provided with the actual Sun Forte C++ compilers are discussed. First of all, the Sun Forte Compilers are commercial program products. So, in contrast to all other interval tools discussed in this paper you have to buy it. The goal of Sun's interval support in C++ is to stimulate development of commercial interval solver libraries and applications by providing program developers with quality interval code, narrow-width interval results, rapidly executing interval code and an easy-to-use software development environment [14].

The following interval extensions are included:

- `interval` template specializations for intervals using `float`, `double`, and `long double` scalar data types (with full support and tuned for speed only for data type `double`).
- extended `interval` arithmetic operations and mathematical functions that form a closed mathematical system: For any possible operator-operand combination, including division by zero and other indeterminate forms involving zero and infinities valid `interval` results are produced. (The empty set is also an interval.)
- Three types (certainly, possibly and set) of interval relational functions like. The certainly relational functions are true if the underlying relation (e. g. less than) is true for every element of the operand intervals. The possibly relational functions are true if any element of the operand intervals satisfy the underlying relation and the set relational functions are true if the interval operands satisfy the underlying relation in the ordinary set theoretic sense (e. g. any interval is set-equal to itself, including the empty interval).
- `interval`-specific functions like `inf`, `sup`, `mid` (midpoint), `isempty`, `intersect`, `disjoint`, `in`, ...
- input and output of intervals also in a special single-number form (the last displayed digit is used to determine the interval's width).

The most exciting feature of Sun's interval support is the possibility of exception free interval computations (containment computations) [15, 16]. Using so called containment sets (set of values that a function can produce when evaluated on the boundary

[1]Using the current version MATLAB 6 Release 12 you have to modify the corresponding M-file: `continue` is a key word and can no longer be used as a name for a variable, so you have to rename this variable

of, or outside its domain of definition in the common mathematical sense must be incorporated) allows valid results (including the empty set and intervals with infinities as bounds), no matter what the value of a function's arguments or an operator's operands. More precisely, the containment set of a function f with respect to an (extended) interval argument $X \subseteq I\!R \cup \{-\infty\} \cup \{\infty\} =: \mathbb{R}^*$ is the closure of the range including all limits and accumulation points, i. e. the set

$$\{f(x) | x \in X \cap D_f\} \cup \{\lim_{D_f \ni x_k \to x^*} f(x_k) | x^* \in X\} \subseteq \mathbb{R}^*$$

If the argument lies strictly outside the natural domain D_f of the function, the result is the empty set (empty interval). Some examples of containment computations are $\log[-1,1] = [-\infty, 0]$, $\sqrt{[-1,1]} = [0,1]$, $\log[-2,-1] = \emptyset$, $\coth[-1,1] = I\!R^*$. The following sample program performs some containment computations:

```
//To compile and link:  CC -xia <progname>
#include <iostream>
#include <suninterval.h> //header file for intervals

//Simplify instantiation of intervals
typedef SUNW_interval::interval<float> interval;

using std::cout;
using std::endl;

int main()
{
   interval x("[-1.5, 3]");
   cout << "x=            " << x << endl;
   cout << "cos(x)=       " << cos(x) << endl;
   cout << "log(x)=       " << log(x) << endl;
   cout << "atan(log(x))= " << atan(log(x)) << endl;
   cout << "log([-2,-1])= " << log(interval(-2,-1)) << endl;
   cout << "[1]/[0]=      " << interval(1)/interval(0) << endl;
   return 0;
}
```

The generated output is:

```
x=            [-.15000000E+001,0.30000000E+001]
cos(x)=       [-.98999250E+000,0.10000000E+001]
log(x)=       [      -Infinity,0.10986124E+001]
atan(log(x))= [-.15707964E+001,0.83235294E+000]
log([-2,-1])= [EMPTY                          ]
[1]/[0]=      [      -Infinity,       Infinity]
```

As a more powerful application we consider the problem of root finding. The interval Newton Method in combination with a bisection process is used to compute enclosures of all roots of a univariate continuously differentiable real valued function.

```
// Interval Newton method using bisection to avoid division by Intervals
// containing zero in the Interval Newton operator.
// Containment computations are performed so no exceptions are raised
// even for interval arguments (partially) outside the natural domain
// of functions and operators.

#include <suninterval.h>
#include <values.h>
#include <iostream>

using std::cout;
using std::endl;

using namespace SUNW_interval;

// Simplify instantiation of intervals with bounds of type double
typedef interval<double> I;

// Internal representation of +00, maybe there is a header file
double Infinity()
{
   return sup(I(1)/I(0));
}

// Data type for univariate interval functions
typedef I (*function) (const I&);

void inewton(function f, function df, const I& x)
{
   static double maxWidth(1e-5);
   if ( !in(0.0, f(x)) ) return; // Definitely no root
   double midx(mid(x));
   if ( midx == -Infinity() )    // midx == -00?
   {
      midx= -MAXDOUBLE;
   }
   else if( midx == Infinity() ) // midx == +00?
   {
      midx=  MAXDOUBLE;          // Set midx to largest finite number
   }
   I fmidx(f(I(midx))), dfx(df(x));
   if ( in(0.0, dfx)          // Avoid 0 in denominator interval
        && wid(x) > maxWidth) // Splitting only if x is still too wide
   {  // Split interval
      inewton(f, df, I(x.inf(), midx));  // Left part
      inewton(f, df, I(midx, x.sup()));  // Right part
      return;
   }
```

```
   I xNew;
   xNew= intersect(I(midx) - fmidx/dfx, x); // Interval Newton operator
   if (xNew==I("[empty]")) return;  // Definitely no root
   if (in_interior(xNew, x))        // Verification ok, one simple root
   {
      cout << "*** Verified " << xNew << endl;
      return;
   }
   if (xNew==x)  // No further improvement
   {
      cout << "Possibly containing a zero: " << xNew << endl;
      return;
   }
   else // One bound improved, try further improvement to get validation
      inewton(f, df, xNew);
}

I pol(const I& x) // Function pol
{
  return (x-I(1))*(x+I(2))*(x-I(3));  // I() interval constructor
}

int main()
{
   I searchRange("[-inf, inf]");    // Entire real line!
   cout << "All roots of (x-1)*(x+2)*(x-3) in the range \n"
        << "   " << searchRange << ":" << endl;
   inewton(pol, dpol, searchRange); // Interval Newton with bisection

   return 0;
}
```

Running the program produces the following output:

```
All roots of (x-1)*(x+2)*(x-3) in the range
   [              -Infinity,               Infinity]:
*** Verified [-.2000000000000000E+001,-.2000000000000000E+001]
*** Verified [0.1000000000000000E+001,0.1000000000000000E+001]
*** Verified [0.3000000000000000E+001,0.3000000000000000E+001]
```

Note, that the complete (unbounded) real line is searched for zeros.

Of course, the program may be improved considerably. Some hints can be found in [9]. But already this simple version allows to attack functions with singularities (e. g. 1+sin(1/x)) and functions with restricted natural domains (e. g. sqrt(1/(x-4)). No exceptions are raised and no roots (whether simple or multiple) are lost. Containment computations are a powerful tool avoiding special programming efforts of various case distinctions.

Up to now no so called interval problem solving routines are available. For example, there are no automatic differentiation package, no linear and nonlinear system solvers,

and no library with vector/matrix data types and operations. Also typical functions like `isempty`, `ispoint`, `succ`, ... and predefined constants are missing.

4 The Interval Library filib++

`filib++` is an extension of the interval library `filib` originally developed in Karlsruhe [2]. The most important aim of the latter was the fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary function. `filib++` extends this library in two aspects. First, it adds a second mode, the extended mode, that extends the exception-free computation mode using special values to represent infinities and Not-a-Number known from the IEEE floating-point standard 754 to intervals. In this mode containment sets are computed to enclose the topological closure of a range of a function defined over an interval [15, 16]. Second, the new state of the art design uses templates and traits classes in order to get an efficient, easily extendable and portable library, fully according to the C++ standard [4].

In contrast to Sun's C++ compilers with interval support `filib++` is freely available. Completely coded in C++ and fully according to the C++ standard the library can be used with many compilers on a large variety of computer platforms.

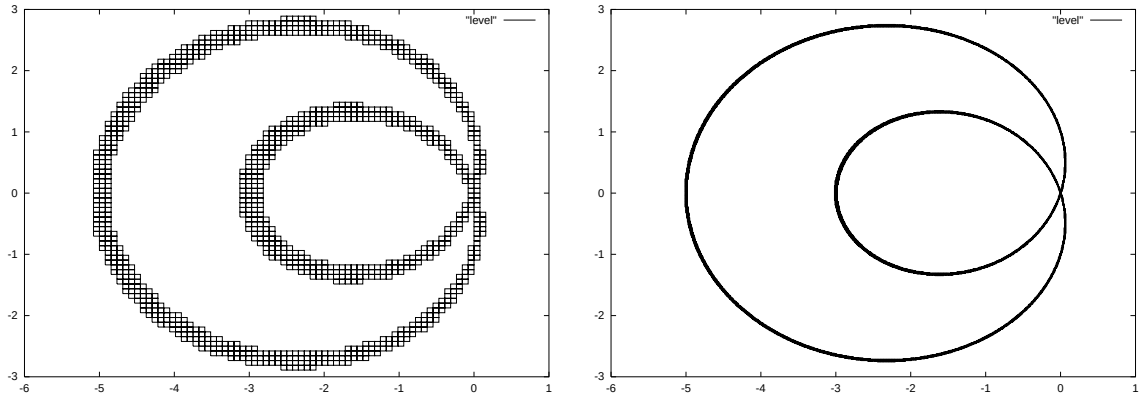
The following program computes enclosures for level curves of two dimensional functions. `filib++` in combination with the standard template library is used. The numerical output is stored in a file called `level`. After completion of the program this file contains data in such a way that gnuplot can display with confidence a graph of the desired level curve using the command `plot "level" with lines`.

```
#include <fstream>
#include <list>

#include <interval/interval.hpp>

using std::cout;
using std::endl;
using std::list;
using std::pair;
using std::make_pair;

//Simplify instantiation of intervals
typedef filib::interval<double> I;

//Simplify access to methods of class fp_traits<>
typedef filib::fp_traits<I::value_type> traits;

pair<I,I> bisect(I x)
{
   //...
}

void levelCurve(I(*f)(I,I), I x, I y, double level, double epsilon)
```

```
{
   //Store boxes containing points of level curve in file "level"
   std::ofstream out("level");
   typedef pair<I,I> rectangle;
   list<rectangle> toDo, done;
   toDo.push_back(make_pair(x,y));
   while( !toDo.empty() )
   {
      rectangle box= toDo.front();
      toDo.pop_front();
      I fRange= f(box.first, box.second);
      if ( in(level, fRange) ) //Box may contain points of level curve?
      {
         if ( width(box) < epsilon ) //Box sufficiently small?
         {  //plot the box possibly containing points of the level curve
            //more precisely: write data points for a plot using gnuplot
            //gnuplot command: plot "filename" with lines
            out << inf(box.first) << " " << inf(box.second) << endl;
            //...

         }
         else
         {
            if( width(box.first) > width(box.second) )
            {
               pair<I,I> p = bisect(box.first);
               toDo.push_back(make_pair(p.first,  box.second));
               toDo.push_back(make_pair(p.second, box.second));
            }
            else
            {
               pair<I,I> p = bisect(box.second);
               toDo.push_back(make_pair(box.first, p.first));
               toDo.push_back(make_pair(box.first, p.second));
            }
         }
      }
   }
   out << endl;
}

I f(I x, I y) //Two dimensional (interval) function
{
   I xx=sqr(x), yy=sqr(y);
   return sqr(xx + yy  + I(4)*x) - xx - yy;
}

int main()
```

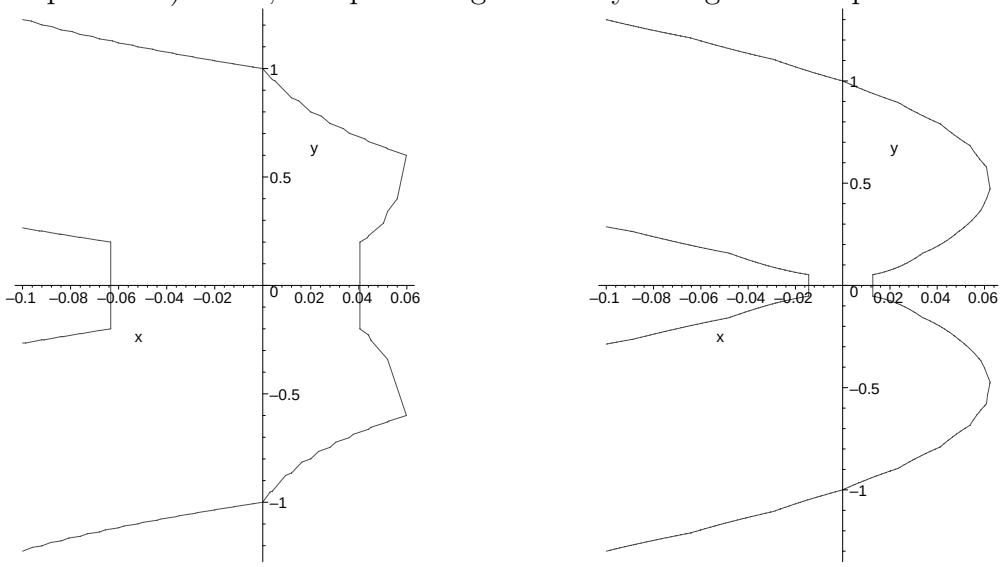
```
{
   traits::setup(); //Do initializations for filib++
   double epsilon=0.1;
   double level=0.0;
   I::precision(16);
   I xRange(-10, 10), yRange(-10, 10);
   cout << "xRange: " << xRange << endl << "yRange: " << yRange << endl;
   levelCurve(f, xRange, yRange, level, epsilon); //Create data for gnuplot
   return 0;
}
```

The figures show the level curve to level 0 for the function $f(x, y) = (x^2+y^2+4x)-x^2-y^2$ using the values 0.1 (left figure) and 0.01 (right figure) for the `epsilon`-parameter.

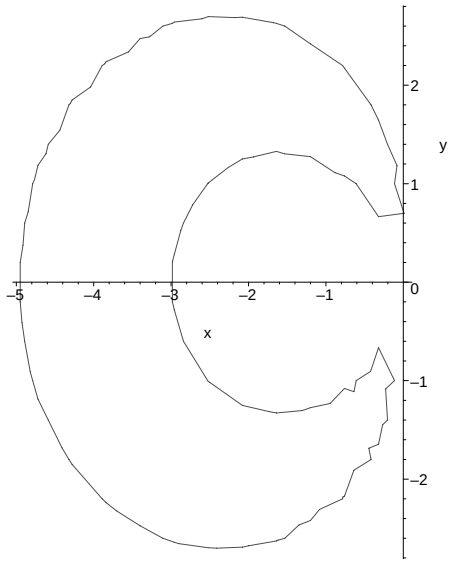


In both cases the graph is covered with confidence by the rectangles shown.

To demonstrate that such plots are not trivial the following figures show the result coming from Maple (the computer algebra system Maple is famous for its great graphical capabilities). Here, interpolation goes totally wrong near the point of intersection.



The left figure uses the default value for the `numpoints` parameter of the `implicitplot` command, the right picture was generated with `numpoints`=9000. In both cases the x-range was from -0.1 to 0.1. Using the x-range from -10 to 1 together with `numpoints` default value we get



Such results can be avoided using interval techniques.

Only small modifications in the source code are necessary to transform the program into source code accepted by Sun's Forte C++ compilers with interval support or accepted by the C++ class library C-XSC to be described in the following section.

5 C++ Class Library C-XSC

C – XSC is a tool for the development of numerical algorithms delivering highly accurate and automatically verified results. It provides a large number of predefined numerical data types and operators of maximum accuracy. These types are implemented as C++ classes. Thus, C – XSC allows high-level programming of numerical applications in C and C++. The C – XSC package is available for many computers with a C++ compiler conformant to the C++ standard [4, 13]. The sources of the new version C – XSC 2.0 are freely available.

The most important features of C – XSC are real, complex, interval, and complex interval arithmetic with mathematically defined properties; dynamic vectors and matrices; dotprecision data types (accurate dot products); predefined arithmetic operators with highest accuracy; standard functions of high accuracy; dynamic multiple-precision arithmetic and standard functions rounding control for I/O data; additional library of problem-solving routines (C++ Toolbox for Verified Computing [1]).

The just mentioned additional library covers the one-dimensional problems

- Accurate evaluation of polynomials
- Automatic differentiation
- Nonlinear equations in one variable
- Global optimization
- Accurate evaluation of arithmetic expressions

- Zeros of complex polynomials

as well as the multi-dimensional problems

- Linear systems of equations
- Linear optimization
- Automatic differentiation for gradients, Jacobians, an Hessian
- Nonlinear systems of Equations
- Global optimization

Further C−XSC packages are freely available or under construction. E. g.

- Verified integration of regular and singular integrals (quadrature and cubature) [17]
- Computation of enclosures for Taylor coefficients of analytical functions
- Slope arithmetic
- Taylor arithmetic
- Initial value problems in ordinary differential equations[2]

So, C−XSC provides a large number of freely available high quality numerical software tools with automatic result verification. The complete source codes as well as a lot of documentation are freely available. See `http://www.math.uni-wuppertal.de/~xsc/`.

6 Conclusion

The great advantage of the interactive programming tool INTLAB and the C++ class library C−XSC is the availability of many problem solving routines (the source codes of the tools itself as well as the sources of the problem solving routines are public and for free). Such routines are not shipped with Sun's (commercial) C++ compilers. Sun replaces the traditional interval computations by so called containment computations. As discussed, containment computations do not rise exceptions. This fact may be stressed advantageously when writing software for parallel and vector computers. filib++ also offers an extended interval mode for containment computations. The design of filib++ is up to date (template programming using traits classes) and the source code of the library is freely available. An automatic differentiation package and an additional package providing a slope arithmetic is under construction.

Using the underlying MATLAB functionality INTLAB handles sparse interval matrices, a feature that is not supported by any other tool considered in this paper. Also midpoint-radius representations of intervals are allowed. Of course, to run INTLAB you must have (must buy) MATLAB.

[2] Available from R. Lohner

A lot of other (more or less specialized) software tools in the field of validated numerics are available. Links may be found on the web. Whenever you have to compute the solution to a numerical problem rigorously verified to be correct do not reinvent the wheel but have a look on software packages already available. For help you can send an email to `reliable_computing@interval.louisiana.edu`.

References

[1] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *C++ Toolbox for Verified Computing*. Basic Numerical Problems. Springer-Verlag, Berlin, 1995.

[2] Hofschuster, W., Krämer, W.: `filib`-*Sources*, `http://www.math.uni-wuppertal.de/org/WRSWT/software.html`, 1998.

[3] Hofschuster, W., Krämer, W., Wedner, S., Wiethoff, A.: *C-XSC 2.0 - A C++ Class Library for Extended Scientific Computing*, Preprint 2001/1, Wissenschaftliches Rechnen/Softwaretechnologie, Universität Wuppertal, 2001.

[4] ISO/IEC 14882: *Standard for the C++ Programming Language*, 1998.

[5] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: *C–XSC – A C++ Class Library for Scientific Computing*. Springer-Verlag, Berlin, 1993.

[6] Krämer, W., Wolff von Gudenberg, J. (eds.): *Scientific Computing, Validated Numerics, Interval Methods*, Kluwer Academic/Plenum Publishers, 2001.

[7] Kulisch, U., Lohner, R., Facius, A.(eds.): *Perspectives on Enclosure Methods*, Springer-Verlag/Wien, 2001.

[8] Lerch, M., Wolff von Gudenberg, J.: *filib++, Specification, Implementation, and Test of a Library for Extended Interval Arithmetic*, RNC4 proceedings, April 2000.

[9] Lerch, M., Tischler, G., Wolff von Gudenberg, J., Hofschuster, W., Krämer, W.: *The Interval Library* `filib++ 2.0`*, Design, Features and Sample Programs*, Preprint 2001/4, Wissenschaftliches Rechnen/Softwaretechnologie, Universität Wuppertal, 2001.

[10] The MathWorks, Inc. (Publisher): *MATLAB, The Language of Technical Computing*, 2001.

[11] Rump, S., M.: *INTLAB - INTerval LABoratory*, in Csendes, T. (Ed.), Developments in Reliable Computing, KluwerAcademic Publisher, Dordrecht, pp. 77-104, 1998.

[12] Rump, S., M.: *Fast and Parallel Interval Arithmetic*, Bit, Vol. 39, No. 3, pp. 534-554, Sept. 1999.

[13] Stroustrup, B.: *The C++ Programming Language*. Special Edition, Addison-Wesley, Reading, Mass., 2000.

[14] Sun Microsystems: *C++ Interval Arithmetic Programming Reference (Forte Developer 6 update 2)*, Sun Microsystems, July 2001.

[15] Walster, G.W.: *Closed Interval Systems*, Technical Report, Sun Microsystems, August 1999.

[16] Walster, W.G., Hansen, E.R., Pryce J.D.: *Extended Real Intervals and the Topological Closure of Extended Real Relations*, Technical Report, Sun Microsystems, Februar 2000.

[17] Wedner, S.: *Verifizierte Bestimmung singulärer Integrale - Quadratur und Kubatur*. Dissertation, Universität Karlsruhe, 2000.