ALBANIAN JOURNAL
OF MATHEMATICS
Volume 4, Number 4, Pages 123–133
ISSN 1930-1235: (2010)

VERIFICATION METHODS AND SYMBOLIC COMPUTATIONS

WALTER KRÄMER

Abstract. Our intpakX package extends the computer algebra system Maple. It allows, e.g., verified numerical calculations (computer-assisted proofs) built on arbitrary precision interval operations. Up to now, only the basic operations are supported in a guaranteed way. Concerning higher mathematical functions supported in Maple, there are no data about their accuracies available/published. Thus, it is not possible or at least very hard to build arbitrary precision interval functions using Maple's intrinsic mathematical functions (nevertheless, intpakX offers such function implementations using some guard digits in an experimental way, which - of course - is not really a reliable mathematical approach).

On the other hand there are software packages supporting reliable multiple precision interval functions like C-XSC, the MPFR and the MPFI libraries, and others. In the talk we discuss the features of some of these libraries in detail. We emphasize the different approaches (arbitrary precision arithmetic, staggered correction arithmetic, functions only for real arguments, functions for complex arguments, ...) and the most important resulting properties of the corresponding implementations. We also compare their performance and we comment on the actual integration of several of these libraries in C-XSC. The missing step is to bring together C-XSC and computer algebra packages like Maple and Mathematica. Combining fast verification methods and symbolic computations deeply extends the range of applications of rigorous mathematical methods.

Key words: Computer-assisted proofs, self-verifying methods, arbitrary precision, interval functions, intpakX, C-XSC.

1. Introduction and general remarks on computer-assisted proofs

It is well known that symbolic computations often suffer from exponential growth of formula strings (memory) and computing time consumption [9]. In many cases it is of great advantage to be able to circumvent this behaviour by applying self-validating numerical methods. The result of such methods are proved to be rigorous in the mathematical sense. These methods are based on the validity of mathematical theorems. Using e.g. interval computations, sufficient conditions for the validity of the mathematical theorems may be verified by the computer itself.

Let us give an example. Brouwer's fixed point theorem may be stated as follows: if a nonempty, convex, compact set X in $I\!R^n$ is mapped by a continuous function $f : I\!R^n \longrightarrow I\!R^n$ into itself, this function has at least one fixed point $x^\star \in X$.

Typically, machine intervals are boxes in $I\!R^n$ with sides parallel to the axes [2, 12, 4, 8]. Such boxes are easily representable (e.g. using an infimum-supremum representation) and they are by their definition convex and compact. Now let F be

kraemer@math.uni-wuppertal.de.

an interval enclosure for the function f representable on the computer (e.g. replace all real operations and elementary function calls by the corresponding machine interval operations/functions). Such an enclosure allows the machine computation of sets containing the range of f over set-valued arguments, typically over machine intervals. Let X be a box (a convex and compact machine interval vector) and let $F(X)$, the result of the machine interval computation, be a subset of X. Then it holds $f(x)|x \in X \subseteq F(X) \subseteq X$. That means, we have proved by some interval machine computations that the continuos function f maps the (convex and compact) interval vector X into itself. The result of the computation assures that Brouwer's fixed point theorem is applicable in the concrete situation and it follows that there exists at least one fixed point x^* of f in X. We see, $F(X) \subseteq X$ can be verified by machine computations and the validity of this relation is sufficient for $\{f(x)|x \in X\} \subseteq X$. Possible conversion errors and rounding errors are captured by machine interval operations (worst case outward rounding). Possible overestimations due e.g. to some kind of wrapping effects or data dependencies do not invalidate the final result (of course, overestimations should be avoided as far as possible to allow $F(X) \subseteq X$ (if the overestimation in the computation of $F(X)$ is too large, this relation does not hold).

2. Newton method to find the zero of a function

Let us consider the simplest case of Newton's method to compute a zero of a continuosly differentiable function g in one real variable. To find the nth root $\sqrt[n]{a}$ of $a \in I\!R_+$ we proceed as follows:

Let

$$g(x) := x^n - a \tag{1}$$

with $g'(x) = nx^{n-1}$. Defining

$$N(x) = x - \frac{g(x)}{g'(x)}\,, \tag{2}$$

the classical Newton iteration computes the iterates

$$x_{k+1} = N(x_k),\ k = 0, 1, 2, \ldots \tag{3}$$

starting from a given initial value x_0.

2.1. Symbolic computations and rational arithmetic.

We start the Newton iteration (3) for the function (1) with fixed values $n = 5$, $a = 32$ and with the rational starting value $x_0 = 1$ Then, obviously, all iterates x_k are rational numbers, i.e. they can be computed error-free using Maple's [11] rational arithmetic. In our case even the answer, i.e. the zero $\sqrt[n]{a}$ of g, is a rational number. What follows is the actual Maple code:

```
> restart;
  g := proc (x) options operator, arrow; x^5-32 end proc;
  dg := unapply(diff(g(x), x), x);


        5
x -> x  - 32


          4
```

```
x -> 5 x

> N := proc (x) options operator, arrow; x - g(x)/dg(x) end proc;

         g(x)
x -> x - -----
         dg(x)

> N := unapply(simplify(N(x)), x);

       / 5    \
     4 \x  + 8/
x -> ----------
           4
        5 x

> xk := 1;
  printf("x0: "); print(xk, 1.0*xk);
  printf("%c", "\n");
  for k to 8 do
    xk := N(xk);
    nodd := ceil(log10(op(1, xk)))+ceil(log10(op(2, xk)));
    printf("Number of decimal digits to represent x%d: %d  %c", k, nodd, "\n");
    if nodd < 100 then print(xk, 1.0*xk) else print(1.0*xk) end if
  end do;

x0:
                                   1, 1.0

Number of decimal digits to represent x1: 3
                               36
                               --, 7.200000000
                               5
Number of decimal digits to represent x2: 14
                            7561397
                            -------, 5.762381497
                            1312200

Number of decimal digits to represent x3: 69
              24748945784387888557877390133166757
              -----------------------------------, 4.615709793
              5361893813970227432939420867060250

Number of decimal digits to represent x4: 345
                                 3.706668058

Number of decimal digits to represent x5: 1722
                                 2.999237997
```

```
Number of decimal digits to represent x6: 8607
                                   2.478483071

Number of decimal digits to represent x7: 43032
                                   2.152390479

Number of decimal digits to represent x8: 215154
                                   2.020104202
```

To represent x_8 exactly as a rational number, already 215154 figures are necessary. However, as an approximation to the value $\sqrt[5]{32} = 2$, $x_8 = 2.0201\dots$ is only accurate to two decimals! The length of the numerator expands by a factor of about 5 at each Newton step. Due to computing time and memory restrictions, the method is obviously not appropriate to compute more than the first few iterates.

2.2. **Interval Newton method using (arbitrary precision) interval operations.** To compute the nth root of the value $a \in I\!R_+$ using an interval Newton method [4] we first introduce the so called Interval-Newton-operator

$$N(X) = N(X, y) := y - G(y)/G'(X) .$$

Here X denotes a closed real interval and y any point in X, e.g. the midpoint of X. If there is a root of g in X then this root is also contained in $N(X, y)$ (if $N(X, y)$ is computable at all). This may be shown by the Mean-Value theorem. The Interval-Newton-operator does not lose a zero of g contained in X. The capital letters G and G' emphasize that we need interval enclosures [4] of the corresponding real valued functions g and g', respectively. We start the interval iteration with starting interval $X_0 := [1/a, a]$ (this interval, and thus all iterates X_k, contain the nth root of a. Also the initial value $x_0 = 1$ of the rational iteration is contained in this starting interval.). The interval Newton method computes the nested sequence of intervals

$$X_{k+1} = N(X_k, \text{midpoint}(X_k)) \,\cap\, X_k,\ k = 0, 1, 2, \dots$$

Let us again set $a = 32$ and $n = 5$. The following Maple code using our Maple Power Tool intpakX ([7, 3, 10] allows e.g. arbitrary precision interval computations) is slightly modified by hand to make it more compact.

Please note, that we use Maple's symbolic manipulation capabilities to automatically generate the first derivative $dg()$ of the function g. The `inapply` command is part of the intpakX package. It transforms a Maple function/expression into an interval function (this means basically that real quantities and real operations are replaced by corresponding interval enclosures and interval operations). This allows to compute verified range enclosures of the original real-valued Maple function/expression over intervals. `mid` indicates the midpoint and `&intersect` denotes an operator computing the intersection of its two interval operands.

```
> restart;
  libname := "/home/kraemer/projekte/braun/master", libname;
  with(intpakX);
> n := 5: a := 32.0:
  g := proc (x) options operator, arrow; x^n-a end proc:
  dg := unapply(diff(g(x), x), x):
```

```
> x := 'x':
  N := inapply( mid(x) - g(mid(x))/dg(x), x ): #Newton operator

> Digits := 40:

> xk := construct(1/a, a); #contains the root of g()
  printf("x0:"); print(xk);
  for k to 11 do  #perform some interval Newton steps
    xk := `&intersect`(N(xk), xk);
    printf("x%d:  %c", k, "\n"); print(xk);
  end do;

x0:
               [0.03125000000000000000000000000000000000000,
          32.00000000000000000000000000000000000000]
x1:
               [0.03125000000000000000000000000000000000000,
                  15.81465263180343736593158610048703849317]
x2:
               [0.03125000000000000000000000000000000000000,
                  7.823231622893632549273354435370367809400]
x3:
               [0.03125000000000000000000000000000000000000,
                  3.879069797905080184045481815927219264884]
x4:
                 [1.958189629694801499712260701420155717296,
                  3.879069797905080184045481815927219264884]
x5:
                 [1.958189629694801499712260701420155717296,
                  2.759821544709750658290262536064053492103]
x6:
                 [1.958189629694801499712260701420155717296,
                  2.217470870974655844998487492795344826269]
x7:
                 [1.983483873581623112376918265014760651179,
                  2.024375423351402189923784960702237493604]
x8:
                 [1.999851475773569004667291861327884059748,
                  2.000171135238750288302281506548692080610]
x9:
                 [1.999999996513240144452564236299207708070,
                  2.000000003740941910738774185453415335888]
x10:
                 [1.999999999999999999097576080686540639748,
                  2.000000000000000000934728169601007961380]
x11:
                 [1.999999999999999999999999999999999999969,
                2.0000000000000000000000000000000000000031]
```

The value of the exact zero $\sqrt[5]{32}$ of g is the integer value 2. Thus, looking at the number of nines behind the leading 1 of the lower bound or at the number of zeros behind the leading 2 of the upper bound allows to see the quadratic convergence property of the Newton iteration. The number of correct digits is typically doubled in each additional step. Using arbitrary precision interval operations (manipulating the `Digits` variable) allow the efficient computation of enclosures of arbitrary accuracy. With this respect, arbitrary precision interval operations may be much more useful than rational operations (see the end of Subsection 2.1).

3. Additional remarks on the interval Newton method and computer-assisted proofs

In Section 2.2 we started the interval Newton iteration with an interval containing the correct answer (this was not checked by the computations but has been verified a priori analytically). But the interval Newton method also allows to check a sufficient criterion with the help of the computer: It is easy to show that if $N(X) \subseteq X$, then the interval X contains exactly one zero of f. The proof is based on Brouwer's fixed point theorem and the fact that $F'(X)$ does not contain 0 (otherwise a division by zero occurs, i.e. $N(X)$ is not defined for the argument X). Thus, the set of derivatives does not contain zero which means that the function g is monotone on X guaranteeing the uniqueness of the zero. The validity of $N(X) \subseteq X$ checked by the computer is a computer-assisted proof that the function f has exactly one zero in the interval X.

A computer-assisted proof for the fact that g has no zero in X results from $N(X) \cap X = \emptyset$. An empty intersection as a result of the corresponding machine interval operation is, again, sufficient for the assertion on g. And again overestimation in the computation of $N(X)$ by (machine) interval operations does not invalidate the sufficiency of the criterion result.

Maple and other computer algebra packages like Mathematica [23], MuPad etc. provide very well designed mathematical function implementations. But to the authors knowledge, the results are not guaranteed to be accurate to the last bit. Even worse, there are now guaranteed error bounds for the function implementation available/published. The function implementations also do not allow interval or complex interval arguments. `intpakX` guarantees interval computations based on the basic arithmetical operations. Computing mathematical functions on intervals is possible, but the realization of these interval functions is based on Maples point functions (accuracy not specified) just using some guard digits. This approach often gives correct results, but of course, it is not really reliable in a mathematical sense. To overcome this unpleasant situation we suggest to combine computer algebra packages with appropriate interval libraries/environments. In the following section we have a look on interval packages supplying their users with different kinds of interval mathematical functions.

4. Some multiple-precision and arbitrary-precision interval packages

There are only a few interval packages supporting multiple-precision interval mathematical functions available [19, 20, 18, 13, 8]. One C++ library with a rather complete set of elementary mathematical functions (trigonometric functions and their inverses, hyperbolic functions and their inverses, exponentials, logarithms, power functions, as well as some other functions) for real and complex machine intervals is C-XSC [8, 5, 6]. C-XSC offers the intrinsic interval data types `interval`, `cinterval` for real and complex intervals with IEEE double numbers as bounds and `l_interval` and `l_cinterval` for staggered precision real and complex intervals. There is also a C-XSC interface to the MPFR and MPFI libraries available. In this case the arbitrary precision data types are called `MPFRClass` and `MPFIClass`, respectively. However, the MPFR and MPFI libraries do not support complex intervals. An additional package to C-XSC is also available delivering staggered precision real and complex intervals with extremely wide exponent range. The data types are called `lx_interval` and `lx_cinterval`, respectively. These staggered data types [16] are based on unevaluated sums of IEEE double numbers. They typically allow precisions up to a several hundred decimal digits.

We first use a variable of the basic complex interval data type `cinterval` to compute an enclosure of the set $\ln(\sin(z))|z \in Z$ with $Z = [0, 1] + i[2, 3] \subset C$. Here Z denotes the rectangle with sides parallel to the axes and with lower left corner (0,2) and upper right corner (1,3). We want to compute a corresponding rectangle, again with sides parallel to the axes, containing the range of the sine function on X. Note, that the shape of the set $\{\ln(\sin(z))|z \in Z\}$ itself is more complex.

The C-XSC source code is as follows:

```
#include <iostream>
using namespace std;
#include <cinterval.hpp>   //complex interval operations
using namespace cxsc;

int main() {
  cinterval z(interval(0,1),interval(2,3)); //complex interval data type
  //complex interval [0,1] + i*[2,3]

  cout << "z: " << endl << z << endl;

  cout << "Enclosure of ln(sin(z)): " << endl << ln(sin(z)) << endl;
}
```

Running the program results in the following output:

```
z:
([  0.000000,  1.000000],[  2.000000,  3.000000])
Enclosure of ln(sin(z)):
([  0.672740,  2.574116],[  0.227314,  1.570797])
```

The computed result `[0.672740, 2.574116] + i*[0.227314, 1.570797]` is guaranteed to be an enclosure of the range of values $\{\ln(\sin(z))|z \in Z\}$.

Let us compute $\ln(\sin(1 + 3i))$ to about 45 good decimals using the staggered precision data type `l_cinterval`.

```
//...as in the listing above
#include <l_cinterval.hpp>   //staggered precision complex intervals

int main() {
  //the global C-XSC variable stagprec allows to control
  //the precision of staggerd-precision quantities:
```

```
  stagprec= 3; //about three-fold double precision

  cout << SetDotPrecision(50,45); //output format

  //create a complex interval in staggered-precision format
  l_cinterval z(interval(1,1),interval(3,3));
  //z is the complex (point) interval [1,1] + i*[3,3]

  cout << "z: " << endl << z << endl;

  cout << "Enclosure of ln(sin(z)): " << endl << ln(sin(z)) << endl; }
```

The output is

```
z:
([ 1.000000000000000000000000000000000000000000000,
   1.000000000000000000000000000000000000000000000 ],
 [ 3.000000000000000000000000000000000000000000000,
   3.000000000000000000000000000000000000000000000 ])
Enclosure of ln(sin(z)):
([ 2.307886347506494601829631969957200324910727432,
   2.307886347506494601829631969957200324910727433 ],
 [ 0.568544730205705952476918985512935541942302411,
   0.568544730205705952476918985512935541942302412 ])
```

The second displayed complex interval is guaranteed to contain the value $ln(sin(1+3i))$. We see that the enclosure is accurate up to the last digits displayed.

The staggered-precision data is a special kind of a multiple-precision data type lohner93,blom09. The operations are performed on floating-point vectors representing exact sums of floating point values. To this end the so called long accumulator (C-XSC dotprecision data type, [8, 24]) is used. It allows the exact computation of dot products of vectors with floating-point components. Because staggered precision numbers are stored as vectors with floating-point components, the precision of staggered numbers is limited by the exponent range of the underlying floating-point screen.

In the next example we perform some time measurements for arbitrary precision real and interval arithmetics based on the libraries MPFR [17, 22] and MPFI [22]. The MPFR library guarantees the best possible accuracy (exactly rounded results for all rounding modes) with respect to the precision used. The interval functions realized by the MPFI library are based on this feature. They give best possible interval enclosures (if the exact function value is representable, this value is return value of the MPFI function call). The set of interval functions realized in the MPFI library is very limited. Complex interval functions are not at all available.

Representable results are reproduced by the MPFI functions. Let x denote the interval $[2, 4]$. Then $\log 2(x) = [1, 2]$ and $\log 2(1/x) = \log 2([1/2, 1]) = [-2, -1]$.

```
#include <iostream>
#include <interval.hpp>  //C-XSC intervals
#include "mpficlass.hpp" //C-XSC interface to MPFI library
using namespace std;
using namespace MPFI;
using namespace cxsc;
```

```
int main() {
  long int prec= 10;
  MpfiClass::SetDefaultPrecision(prec);  //use prec bit for mantissa
  MpfiClass::SetBase(2);                 //base for input/output

  MpfiClass x(interval(2.0,4.0));        //interval [2,4]
  cout << "log2(x): " << log2(x) << endl;
  MpfiClass r;
  r= 1/x;
  cout << "log2(1/x)=log2(r): " << log2(r) << endl;
  cout << "log10(x): " << log10(x) << endl;
}
```

Running the program produces the following output:

```
log2(x):             [ 1.00000,    1.00000e1 ]
log2(1/x)=log2(r):   [-1.00000e1, -1.00000   ]
log10(x):            [ 1.00110e-2, 1.00111e-1]
```

Note that the interval bounds are printed as binary numbers. The results are as predicted.

The following program is used to do some time measurements for the arbitrary precision MPFI interval functions. We compute enclosures for the sine function at the point interval $[7, 7]$ with 1000 bit starting precision and doubling the precision within a loop until $2^9 \times 1000 = 512000$ bit are reached.

```
#include <interval.hpp>
#include "mpficlass.hpp"
#include "timer.hpp"
using namespace std;
using namespace MPFI;
using namespace cxsc;

int main() {
  double start;
  long int precision=1000; //number of mantissa bits

  for (int i= 0; i<= 9; i++) {
    MpfiClass x(interval(7), precision);  //point interval [7,7]
    cout << precision << " bits,  ";
    start= GetTime();
    sin(x);  //function call
    cout << "time used: " << GetTime()-start << " sec" << endl;
    precision+= precision; //precision doubling
  }
}
```

Running the program produces the following output:

```
  1000 bits,  time used: 0.000250101 sec
  2000 bits,  time used: 0.000307083 sec
  4000 bits,  time used: 0.000962019 sec
  8000 bits,  time used: 0.00355291 sec
 16000 bits,  time used: 0.0135369 sec
 32000 bits,  time used: 0.0501981 sec
 64000 bits,  time used: 0.191191 sec
128000 bits,  time used: 0.698907 sec
256000 bits,  time used: 2.63609 sec
512000 bits,  time used: 8.5943 sec
```

The time needed to compute the sine function using Maple at the point 7.0 to 100000 decimal places (about 332200 binary digits) measured by

```
restart: Digits:=100000; st:= time(): sin(7.0): time()-st;
```

is: `11.605` seconds.

The C-XSC as well as the Maple results have been computed on the same machine.

Maple's sine function implementation seems to be not as efficient as the interval sine function for MpfiClass interval variables in C-XSC. There is no guarantee of good digits in the Maple result whereas the MPFI/C-XSC enclosure is the best possible result (guaranteed by the MPFR and MPFI libraries) with respect to the actual precision setting.

5. Concluding remarks

We urgently need software tools combining symbolic computations and verification methods. There are several promisingly first approaches (see e.g. [14]). However, a lot of further work (theoretical research as well as highly demanding software development) has to be done to get really powerful hybrid methods. The author is confident that it's worth the effort. The outcome will allow the user to do more and more automatized rigorous mathematics on the computer, not only based on symbolic manipulations but also based on very fast floating-point (interval) computations. Also highly sophisticated but often unsafe approximate methods may be complemented by mathematically rigorous supplementations.

References

[1] Weblink to C-XSC
`http://www.math.uni-wuppertal.de/wrswt/xsc/cxsc_new.html`

[2] Alefeld, G., HerzbergerJ.: Introduction to Interval Computations. Academic Press, New York, 1983.

[3] Markus Grimmer.: Interval Arithmetic in Maple with intpakX. PAMM - Proceedings in Applied Mathematics and Mechanics, Vol. 2, Nr. 1, p. 442-443, Wiley-InterScience, 2003.

[4] Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: Numerical Toolbox for Verified Computing I: Basic Numerical Problems. Springer Verlag, 1993.

[5] Hofschuster, W., Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, pp. 15 - 35, 2004.

[6] Hofschuster, W., Krämer, W., Neher, M.: C-XSC and Closely Related Software Packages. Preprint 2008/3, Universität Wuppertal, 2008; published in: Dagstuhl Seminar Proceedings 08021 - Numerical Validation in Current Hardware Architectures, LNCS 5492, Springer-Verlag, pp 68-102, 2008.

[7] `intpakX` link at the University of Wuppertal:
`http://www.math.uni-wuppertal.de/~xsc/software/intpakX/`

[8] Klatte, R., Kulisch, U., Wiethoff, A., Lawo, Chr., Rauch, M.: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993.

[9] Krämer, W.: Accurate Computation of Chaotic Dynamical Systems. In: A. Aggarwal (ed): Proceedings to Mathematics and Computers in Biology and Chemistry (MCBC 07), Vancouver, Canada, pp. 74-79, 2007.

[10] Krämer, W.: intpakX - An Interval Arithmetic Package for Maple. Proceedings of the 12th GAMM-IMACS Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, SCAN 2006, IEEE Computer Society, ISBN 0-7695-2821-X, 2007.

[11] Maplesoft: `http://www.maplesoft.com/applications/app_center_browse.aspx?CID=13&SCID=155`

[12] Neumaier, A.: Interval Methods for Systems of Equations. Encyclopedia of Mathematics and its Applications 37, Cambridge University Press, Cambridge, UK, 1990.

[13] Aberth, O.: Introduction to Precise Numerical Methods. Academic Press, New York, 2007.

[14] Popova, E., Krämer, W., Russev, M.: Integration of C-XSC Automatic Differentiation in Mathematica. Preprint 3/2010, IMI-BAS, Sofia, March, 2010. See `http://www.math.bas.bg/~epopova/papers/10-preprintAD.pdf`

[15] Adams, E., Kulisch, U.: Scientific Computing With Automatic Result Verification. Academic Press, Inc., 1993.

[16] Blomquist, F., Hofschuster, W., Krämer, W.: A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range. Lecture Notes in Computer Science LNCS 5492, pp. 41-67, Springer, 2009.

[17] Fousse, L., Hanrot, G., Lefevre, V., Pelissier, P., Zimmermann, P.: MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding. ACM Transactions on Mathematical Software, Vol.33, No.2, Article 13, 2007.

[18] Grimmer, M., Petras, K., Revol, N.: Multiple Precision Interval Packages: Comparing Different Approaches. In Lecture Notes in Computer Science, Vol. 2991, pp. 64–90, Springer, 2004.

[19] Krämer, W.: Multiple Precision Computations With Result Verification. In [1], pp. 325–356, 1993.

[20] Krämer, W., Kulisch, U., Lohner, R.: Numerical Toolbox for Verified Computing II – Advanced Numerical Problems (draft). Chapter 7, Multiple-Precision Arithmetic Using Integer Operations, pp. 210–251, 1998.
Online availabe, see `http://www.math.uni-wuppertal.de/wrswt/literatur/tb2.ps.gz`

[21] Lohner, R.: Interval Arithmetic in Staggered Correction Format. In [1], pp. 301–342, 1993.

[22] Revol, N. and Rouillier, F.: Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. Reliable Computing, Vol. 11, pp. 275–290, 2005.

[23] Wolfram Research Inc.: *Mathematica*, Version 5.2, Champaign, IL, 2005.

[24] Zimmer, M., Krämer, W., Bohlender, G., Hofschuster, W.: Extension of the C-XSC Library With Scalar Products With Selectable Accuracy. Preprint BUW-WRSWT 2009/4, University of Wuppertal, 2009; in press, Serdica Journal of Computing, 2010.

Walter Krämer, Scientific Computing/Software Engineering, Faculty of Mathematics and Natural Sciences, University of Wuppertal, 42119 Wuppertal, Germany.