C-XSC 2.0
A C++ Library for Extended Scientific Computing

Werner Hofschuster and Walter Krämer

Bergische Universität Wuppertal, Scientific Computing/Software Engineering,
Gaußstraße 20, D-42097 Wuppertal, Germany
{Hofschuster, Kraemer}@math.uni-wuppertal.de
http://www.math.uni-wuppertal.de/~xsc

Abstract. In this note the main features and newer developments of the C++ class library for extended scientific computing C-XSC 2.0 will be discussed.
The original version of the C-XSC library is about ten years old. But in the last decade the underlying programming language C++ has been developed significantly. Since November 1998 the C++ standard is available and more and more compilers support (most of) the features of this standard. The new version C-XSC 2.0 conforms to this standard. Application programs written for older C-XSC versions have to be modified to run with C-XSC 2.0. Several examples will help the user to see which changes have to be done. Note, that all sample codes given in [6] have to be modified to work properly with C-XSC 2.0.
All sample codes listed in this note will be made available on the web page http://www.math.uni-wuppertal.de/~xsc/cxsc/examples.

1 Introduction

For those who are not so familiar with C-XSC let us first motivate the library by quoting essential parts (with slight modifications) from the preface of the book [6]:

The programming environment C-XSC (C++ for eXtended Scientific Computing) is a powerful and easy to use programming tool, especially for scientific and engineering applications. C-XSC is particularly suited for the development of numerical algorithms that deliver highly accurate and automatically verified results. It provides a large number of predefined numerical data types and operators of maximum accuracy. 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 rounding control for the input and output of data.

Accumulation of numbers is the most sensitive operation in floating-point arithmetic. By that operation scalar products of floating-point vectors, matrix

products etc. can be computed without any error in infinite precision arithmetic, making an error analysis for those operations superfluous. Many algorithms applying that operation systematically have been developed. For others the limits of applicability are extended by using this additional operation. Furthermore, the optimal dot product speeds up the convergence of iterative methods (cited from [10, 11]). C-XSC provides accurate dot products via software simulation (hardware support should increase the computation speed by 2 orders of magnitude, again, see [11]). Computing $x*y$ for floating point vectors x, and y in C-XSC results in the best possible floating point result (exact mathematical result rounded to the nearest floating point number). Using the new C-XSC data type `dotprecision` the user can even store the result of dot products of floating point vectors with even millions of components without any error. The so called staggered format allows multiple-precision computations. The realization of arithmetic operations for variables of this data type use extensively the accurate dot product. With approriate hardware support for dot product operations the staggered arithmetic would be very fast.

C-XSC consists of a run time system written in ANSI C and C++ including an optimal dot product and many predefined data types for elements of the most commonly used vector spaces such as real and complex numbers, vectors, and matrices. Operators for elements of these types are predefined and can be called by their usual operator symbols. Thus, arithmetic expressions and numerical algorithms are expressed in a notation that is very close to the usual mathematical notation.

Additionally, many problem-solving routines with automatic result verification (e.g. C++ Toolbox for Verified Computing with one- and multi-dimensional solvers for systems of linear equations, linear optimization, automatic differentiation, nonlinear systems of equations, global optimization and further packages like slope and taylor arithmetic or quadrature and cubature of singular integrals) have been developed in C-XSC for several standard problems of numerical analysis. All software is freely available.

2 Overview on the new version C-XSC 2.0

Due to the following observations older C-XSC programs have to be modified slightly to run with C-XSC 2.0 (for details please refer to paragraph 4):

- All C-XSC routines are now in the namespace `cxsc`. So you have to fully qualify names of C-XSC routines (e. g. cxsc::sin(cxsc::intval(3.0))) or you have to include the line `using namespace cxsc;` in your source code.
- Now typecast constructors are available
- Constant values formerly passed by reference are now passed by `const` references
- Modifications in the field of subvectors and submatrices have been done
- The error handling is now done using the C++ exception handling mechanism (using `try`, `catch`, and appropriate exception classes)
- The new version of the library uses templates extensively

The source code of C-XSC 2.0 is freely available from
`http://www.math.uni-wuppertal.de/~xsc/xsc/download.html` and the source code of a new version of the C++ Toolbox for Verified Computing [1] which works with C-XSC 2.0 is also freely available from the same web site.

3 Freely available software based on C-XSC 2.0

Here we list (additional) software based on C-XSC 2.0 which is freely available from our web-site:

a) (Modified) Toolbox for Verified Computing (see [1]). This toolbox comprises a couple of verification algorithms for one- and multi-dimensional numerical problems:

a1) The available one-dimensional problem solving routines are:

- Accurate polynomial evaluation
- Automatic differentiation
- Nonlinear equations in one variable
- Selfverifying global optimization
- Accurate arithmetical expressions
- Zeros of complex polynomials

a2) The available multi-dimensional problem solving routines are:

- Systems of linear equations
- Linear optimization
- Automatic differentiation (gradient, Jacobi-, Hesse matrix)
- Nonlinear systems of equations
- Global optimization

b) Further available software packages are:

- Interval slope arithmetic (Breuer)
- Interval Taylor arithmetic (Breuer)
- Mathematical functions for complex rectangular intervals (Westphal)
- Verified quadrature and cubature of nonsingular and singular integrals (Wedner, see [8, 20])
- Verified estimates for Taylor coefficients of analytic functions (Neher [16])
- Routines to compute rigorous worst case a priori bounds for absolute and/or relative errors of floating point algorithms (Bantle [7])
- Solvers for under- and overdetermined systems of linear equations (Hölbig [3])
- Verified solutions of ordinary differential equations (Lohner [13])

You can download the source code of all software packages from
`http://www.math.uni-wuppertal.de/~xsc`.
There, you also find more specific information on the packages as well as some preprints.

4 Which modifications in source codes are required?

In this section we try to answer the most frequently asked questions of C-XSC users concerning the migration of older C-XSC application programs to the new C-XSC 2.0 version. For those who are familiar with the C++ standard [5] the source code modifications should be rather obvious (see e.g. Stroustrup [19], Meyers [14, 15]).

To make available the advanced input and output facilities (stream concept) of C++ you must include the headerfile `iostream` using the source line `#include <iostream>`. Note, the name of the header is **not** `iostream.h`. In general, the names of system header files coming with C++ do not have an extension.

To perform conversions of interval constants given as strings C-XSC uses the header file `#include <string>`. This header introduces (dynamic) C++ strings with predefined operators.

C-XSC delivers several header files. The extension of these files is `.hpp`. The header files correspond to the additional numerical data types available in C-XSC (like `interval`, `imatrix`, `cmatrix`, ...). The name of the header files are

`cdot.hpp`	`dot.hpp`	`l_complex.hpp`	`lrvector.hpp`
`cidot.hpp`	`idot.hpp`	`l_imath.hpp`	`real.hpp`
`cimatrix.hpp`	`imath.hpp`	`l_interv.hpp`	`rmath.hpp`
`cinterval.hpp`	`imatrix.hpp`	`l_real.hpp`	`rmatrix.hpp`
`civector.hpp`	`interval.hpp`	`l_rmath.hpp`	`rvector.hpp`
`cmatrix.hpp`	`intmatrix.hpp`	`limatrix.hpp`	
`complex.hpp`	`intvector.hpp`	`livector.hpp`	
`cvector.hpp`	`ivector.hpp`	`lrmatrix.hpp`	

The leading `l` in the name of a header file indicates a long precision (staggered) data type, `dot` indicates dotprecision data types able to store dot products without errors (long accumulators). In contrast to system header files which are included in the form `#include <header>` C-XSC header files are included using double quotes `#include "cxscheader.hpp"`.

The result type of the routine `main()` should be `int`.

Newer C++ compiler implement the namespace concept more strictly. The standard namespace of C++ is called `std`. All C-XSC routines are defined in the namespace `cxsc`. If you don't want to fully qualify the names of such routines (e. g. `std::cout`, or `cxsc::interval`) you should include the two source lines

```
using namespace std;  //make available names like cout, endl, ...
using namespace cxsc; //make available names of C-XSC routines
```

in your application code.

The following simple example program demonstrates most of the points from above. It checks whether the number 0.1 is representable as a point interval in C-XSC. If this is not the case, the decimal number 0.1 is not exactly representable as a double number.

```
#include <iostream>     //C++ stream concept for input and output
#include <string>       //ANSI C strings
#include "interval.hpp" //C-XSC header file for data type interval

using namespace std;    //make available names like cout, endl, ...
using namespace cxsc;   //make available names of C-XSC routines

int main()
{
   interval x; //x is an interval variable

   string("[0.1,0.1]") >> x;  //convert the interval constant to its
                              //internal binary representation
      //(using directed roundings)
   if (Inf(x) != Sup(x))
         cout << "Number x has no exact binary representation!";
      else
         cout << "Number x has an exact binary representation!";

   cout << endl << "x = " << x << endl; //decimal output using
                                        //C++ streams
   cout << Hex  << "x = " << x << endl; //hexadecimal output

   return 0;
}

/* ----------------------- Output --------------------------
Number x has no exact binary representation!
x = [  0.099999,  0.100001]
x = [+19999999999999e3FB,+1999999999999Ae3FB]
------------------------------------------------------- */
```

If your (older) application code contains calls to conversion functions like `_interval(...)` you should now use constructor calls like `interval(...)` instead. The C-XSC conversion functions (starting with an underscore) are obsolete.

Several function signatures of C-XSC routines have been changed from reference parameters (`T& x`) to `const` reference parameters (`const T& x`). The following C++ sample program demonstrates some consequences.

```
#include <iostream>
using namespace std;

void f(const double& x) { cout << "Formal argument with const" << endl; }

void f(double& x) { cout << "No const qualifier" << endl; }

int main()
{
   double x=2;
```

```
   f(1.0);    //1, actual argument is not an lvalue
   f(x);      //2, x is an lvalue
   f(1.0+x);  //3, actual argument is not an lvalue
   f(x+x);    //4, actual argument is not an lvalue
   return 0;
}
/*
Formal argument with const
No const qualifier
Formal argument with const
Formal argument with const
*/
```

Note, due to the `const` qualifier the signatures in the two definitions of f() are different in C++! If we remove the first definition of f(), the function calls in the lines indicated by 1, 3, and 4 produce errors during the compilation process. In these cases the actual arguments are not lvalues whereas the formal argument of type `double&` (see the second definition of f) requires an lvalue.

Note, that the two definitions

```
void g(const double x) {cout << "Formal argument with const" << endl;}
void g(double x) {cout << "No const qualifier" << endl;}
```

are not allowed simultaneously in a C++ program unit. Here, the formal arguments are not declared as references. This implies that in both cases the actual argument in a function call is passed by value (the values of the actual arguments can not be changed in the body of the function). So an additional `const` qualification does not make sense.

Operators like `[]` as member function of a class may be overloaded differently for objects and `const`-objects. This is demonstrated by the following C++ sample code (the `const` between the parameterlist and the body of the operator definition indicates that in the body of the function the attributes of the left hand side object in a corresponding operator call are not modifiable):

```
#include<iostream>
using namespace std;

typedef double T;

struct vector {
   vector(int k) //constructor
   {
      start= new T[k];
      for (int i=0; i<k; i++) start[i]= i;
   }

   //operator [] may be applied to vectors
   //elements are readable and writable (result type is T&)
   T& operator[](int k)
   {
```

```
      cout << "[] without const ... " << endl;
      return  start[k];
   }

   //operator [] may be applied to const vectors
   //elements are only readable (result type is const T&)
   const T& operator[](int k) const
   {
      cout << "[] with const ... " << endl;
      return  start[k];
   }

   ~vector() { delete[] start; } //destructor
 private:
   T* start;
};

int main() {
   vector x(3);
   cout << "x[2]: " << x[2] << endl;
   x[2]= 5; //Note, calling operator[] creates output (see below)
   cout << "x[2]: " << x[2] << endl;
   const vector y(3);  //the same as vector const y(3);
   cout << "y[2]: " << y[2] << endl;
// y[2]= 5; //would lead to a compile time error:
            //The left operand cannot be assigned to
   return 0;
}
/* Output:

x[2]: [] without const ...
2
[] without const ...
x[2]: [] without const ...
5
y[2]: [] with const ...
2
*/
```

In contrast to the older C-XSC versions C-XSC 2.0 uses additional helper classes `intvector_slice`, `rvector_slice`, `ivector_slice`, `cvector_slice`, `civector_slice`, `l_rvector_slice`, `l_ivector_slice`, `intmatrix_slice`, `intmatrix_subv`, `rmatrix_slice`, `rmatrix_subv`, `imatrix_slice`, `imatrix_subv`, `cmatrix_slice`, `cmatrix_subv`, `cimatrix_slice`, `cimatrix_subv`, `l_rmatrix_slice`, `l_rmatrix_subv`, `l_imatrix_slice`, `l_imatrix_subv` to implement subvectors and subarrays.

The following program shows how the first row and the first column of a real matrix may be modified calling a function called `testfct`. The formal parameter of this function must be of data type `rmatrix_subv`.

```
#include <iostream>
#include "rmatrix.hpp" //C-XSC header for real matrices
                       //header for real vectors is included automatically

using namespace std;
using namespace cxsc;

void testfct(const rmatrix_subv& y) //pay attention to the data type of y
//void testfct(const rvector& y) an error message or a warning would be
//                               generated by actual compilers
{
    for (int i=Lb(y); i<=Ub(y); i++) y[i]= i;
}

int main(void)
{
   rmatrix M;          //M is a real matrix
   int dim;
   cout << "Dimension = "; cin >> dim;

   Resize(M,dim,dim);  //create M with dim rows and dim columns
   M= 1;               //set all elements of M to 1

   cout << "Matrix M:" << endl << M << endl;
   testfct(M[1]);      //M[1] means the first row of M
   cout << "Matrix M:" << endl << M << endl;

   testfct(M[Col(1)]); //M[Col(1)] means the first column of M
   cout << "Matrix M:" << endl << M << endl;

   M[Col(1)]= 9;       //set all elements of column 1 to 9
   cout << "Matrix M:" << endl << M << endl;

   return 0;
}

/* Output

Dimension = 3
Matrix M:
  1.000000   1.000000   1.000000
  1.000000   1.000000   1.000000
  1.000000   1.000000   1.000000

Matrix M:
  1.000000   2.000000   3.000000
  1.000000   1.000000   1.000000
  1.000000   1.000000   1.000000

Matrix M:
```

```
  1.000000   2.000000   3.000000
  2.000000   1.000000   1.000000
  3.000000   1.000000   1.000000

Matrix M:
  9.000000   2.000000   3.000000
  9.000000   1.000000   1.000000
  9.000000   1.000000   1.000000
*/
```

5 Examples

In this section we give a couple of complete sample codes to demonstrate the usage and several features of C-XSC 2.0.

5.1 Example: Accurate summation of floating-point numbers

Let us start with a very simple demonstration of how the accurate dot product feature may be used to get accurate results when summing up floating-point numbers of very different orders of magnitude. The C-XSC routine `accumulate(a,x,y)` computes `a+x*y` without any error. Here `x` and `y` are floating-point numbers and `a` is a variable of type `dotprecision` (a so called long accumulator):

```
//Severe cancellation when computing the sum of three numbers
//Using a dotprecision variable results in the correct result

#include <iostream> //C++ input and output
#include "dot.hpp"  //make available C-XSC's accurate dot product feature
using namespace std;
using namespace cxsc; //make available C-XSC names without cxsc::

int main() {
   const real large(1.23e35);    //create a large number

   dotprecision a(0); //a is a dot precision variable initialized by 0
   accumulate(a,  1.0, large);  //a = 1.0*large = 1.23e35
   cout << a << endl;

   accumulate(a,  1.0, 1.5);    //a = 1.0*large + 1.0*1.5
                                //  = 1.2300...015e35
   accumulate(a, -1.0, large);  //a=  large + 1.5 - large = 1.5
   cout << "Final correct result is" << a << endl;

   cout << "Naive floating point evaluation gives" << endl
        << "  the totally wrong result"
        << large + 1.5 - large << endl;
   return 0;
```

```
}
/*  output:
1.2300000000E+0035
Final correct resultis is   1.5000000000
Naive floating point evaluation gives
  the totally wrong result  0.000000
*/
```

The possibility to compute dot products of floating point vectors accurately is the key for the implementation of matrix/vector operations of maximum accuracy in C-XSC. This feature is also used extensively in defect correction steps of iterative schemes. The operations for the staggered data type (multiple-precision) available in C-XSC [9] are heavily based on accurate dot product computations.

5.2 Example: Accurate evaluation of arithmetical expressions

The following arithmetical expression has been used by Loh and Walster [12] as an example in which numerical evaluations using IEEE 754 arithmetic gave a misleading result, even though use of increasing arithmetic precision suggested reliable computation (the expression is a rearrangement of Rump's original example given in [17]). Evaluating

$$f(a,b) = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \qquad (1)$$

for $a = 77617$ and $b = 33096$ using 32-bit, 64-bit, and 128-bit round-to-nearest IEEE-754 arithmetic produces:

32-bit: $f = 1.172604$
64-bit: $f = 1.1726039400531786$
128-bit: $f = 1.172603940053178631858834904520 1838$

However, the correct result is -0.8273960...

To compute a sharp enclosure of $f(a,b)$ we use the staggered interval arithmetic available in C-XSC.

```
#include <iostream>
#include "l_interval.hpp" //staggered intervals (multi-precision intervals)

using namespace cxsc;     //make available routines from namespace cxsc
using namespace std;

l_interval f ( const l_interval& a, const l_interval& b )
{
  l_interval z;           //multi-precision interval

  z = (333.75 - power(a,2))*power(b,6) + power(a,2)*(11.0*power(a,2)
      *power(b,2) - 121.0*power(b,4) - 2.0) + 5.5*power(b,8) + a/(2.0*b);
```

```
  return(z);
}

int main( )
{
  l_real     a, b; //multi-precision reals
  l_interval res;  //multi-precision interval
  real       Eps;

  cout << "Enter the arguments:" << endl;
  cout << "   a = " ; cin >> a;  //read a multi-precision real
  cout << "   b = ";  cin >> b;
  cout << endl;

  cout << "Desired accuracy: Eps = "; cin >> Eps;
  cout << endl;

  cout << "Evaluation of (333.75 -a^2)b^6+a^2(11a^2b^2-121b^4-2)
           +5.5b^8+a/(2b)"
       << endl << endl;

  stagprec=0;
  do {
      stagprec++;
      res = f(l_interval(a),l_interval(b));
      //Output format via dotprecision
      cout << SetDotPrecision(16*stagprec, 16*stagprec-3);
      cout << "Interval enclosure: " << res << endl;
      cout << SetDotPrecision(5,2);
      cout << "Diameter:           " << diam(res) << endl;
  } while (diam(res)>Eps);

  return 0;
}

/* ----------------------- Output --------------------------
Enter the arguments:
   a = 77617
   b = 33096

Desired accuracy:
   Eps = 1e-100

Evaluation of (333.75 -a^2)b^6+a^2(11a^2b^2-121b^4-2)+5.5b^8+a/(2b)

Interval enclosure: [-3.5417748621523E+0021,
                     3.5417748621523E+0021]
Diameter:           7.08E+0021

Interval enclosure: [-6.5534827396059947204776108265OE+0004,
```

```
                     1.17260394005317869492444060598]
Diameter:           6.55E+0004

Interval enclosure: [-0.82739605994682136814116509547981629199903311 6,
                     -0.82739605994682136814116509547981629199903311 5]
Diameter:           2.74E-0048

Interval enclosure: [-0.82739605994682136814116509547981629199903311
                        5843848199178149,
                     -0.82739605994682136814116509547981629199903311
                        5843848199178148]
Diameter:           1.52E-0064

Interval enclosure: [-0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        06,
                     -0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        05]
Diameter:           1.69E-0080

Interval enclosure: [-0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        06
213,
                     -0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        06
212]
Diameter:           1.87E-0096

Interval enclosure: [-0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        06
                        2122310853275320281,
                     -0.82739605994682136814116509547981629199903311
                        5843848199178148416727096930142615421803239
                        06
                        2122310853275320280]
Diameter:           2.08E-0112
```

The last enclosure is accurate to more than 110 digits (that is to all digits printed).

Let us now solve the same problem (1) (example from Rump/Loh & Walster) with the toolbox algorithm for the accurate evaluation of arithmetical expressions:

```
#include <expreval.hpp>     //Expression evaluation

using namespace cxsc;
using namespace std;

Staggered f ( StaggArray& v )
{
  Staggered a, b;
```

```
  a = v[1];
  b = v[2];

  return((333.75 - Power(a,2))*Power(b,6) + Power(a,2)*(11.0*Power(a,2)
    *Power(b,2) - 121.0*Power(b,4) - 2.0) + 5.5 * Power(b,8) + a/(2.0*b));
}

int main ( )
{
  real     Eps, Approx;
  int      StaggPrec, Err;
  rvector  Arg(2);
  interval Encl;

  cout << SetPrecision(23,15) << Scientific;   //Output format

  cout << "Evaluation of (333.75 -a^2)b^6+a^2(11a^2b^2-121b^4-2)
           +5.5b^8+a/(2b)"
       << endl << endl;

  cout << "Enter the arguments:" << endl;
  cout << "   a = " ; cin >> Arg[1];
  cout << "   b = ";  cin >> Arg[2];
  cout << endl;

  cout << "Desired accuracy:  Eps = " ; cin >> Eps;
  cout << endl;

  Eval(f, Arg, Eps, Approx, Encl, StaggPrec, Err);

  if (!Err) {
    cout << "Floating-point evaluation:  " << Approx << endl;
    cout << "Interval enclosure:         " << Encl << endl;
    cout << "Defect corrections needed:  " << StaggPrec << endl;
  }
  else
    cout << EvalErrMsg(Err) << endl;

  return 0;
}

/* ---------------------- Output -------------------------

Evaluation of (333.75 -a^2)b^6+a^2(11a^2b^2-121b^4-2)+5.5b^8+a/(2b)

Enter the arguments:
   a = 77617
   b = 33096

Desired accuracy:  Eps = 1e-15
```

```
Floating-point evaluation:  1.172603940053179E+000
Interval enclosure:       [-8.273960599468215E-001,-8.273960599468213E-001]
Defect corrections needed:  2
------------------------------------------------------------ */
```

Again, the computed interval enclosure is sharp.

5.3 Example: Linear System of Equation

We want to solve the (ill-conditioned) system of linear equations $Ax = b$ with

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix},\ b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},\ x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

The correct solution is $x_1 = 205117922, \quad x_2 = 83739041.$

To solve this 2×2 system numerically we first use the wellknown formulas

$$x_1 = \frac{a_{22}}{a_{11}a_{22} - a_{12}a_{21}}, \quad x_2 = \frac{-a_{21}}{a_{11}a_{22} - a_{12}a_{21}} \tag{2}$$

The following ANSI-C program

```
#include <stdio.h>

int main(void)
{
   double a11= 64919121.0, a12= -159018721.0,
          a21= 41869520.5, a22= -102558961.0,
          h1, h2, x1, x2;

   h1= a11*a22;
   h2= a12*a21;
   x1=  a22/(h1-h2);
   x2= -a21/(h1-h2);
   printf("x1= %15f   x2= %15f\n", x1, x2);
   return 0;
}
```

produces the totally wrong result

$$x_1 = 102558961, \quad x_2 = 41869520.5.$$

I. e. using IEEE double-arithmetic to evaluate the formulas (2) shown above give meaningless numerical results.

We now try to solve the linear system using Matlab.
Here we compute the inverse matrix (theoretically, the first column of the inverse is the solution of the linear system)

```
>> inv(A)
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.651447e-17.
ans =
          106018308.007132         -164382474.017831
  43281793.0017831                 -67108864
>> A*inv(A)
ans =
     0     2
    -1     2
>> inv(A)*A
ans =
     1     2
     0     1
```

A*inv(A) as well as inv(A)*A should give the identity matrix. Obviously, the computed results are again not reliable. But this time we get at least a warning from Matlab.

If we try to compute an enclosure of the solution vector x using Rump's IntLab package [18]

```
x = verifylss(A,b)
```

we get the same warning as in Matlab (indeed it is the Matlab warning) and the output

```
No inclusion achieved.
x =
   NaN
   NaN
```

IntLab is not able to solve the system. No meaningless numerical values are produced.

Let us now try to solve our ill-conditioned problem using C-XSC. Calling the solver for systems of linear equations from the Toolbox library [2] (using the interactive toolbox example program `lss_ex`) we get the following enclosure of the solution:

```
Enter the dimension of the system: 2

Enter matrix A:
64919121 -159018721
41869520.5 -102558961

Enter vector b:
1 0

Naive floating-point approximation:
```

```
 2.051179220000000E+008
 8.373904100000000E+007

Verified solution found in:
[ 2.051179220000000E+008, 2.051179220000000E+008]
[ 8.373904100000000E+007, 8.373904100000000E+007]

Condition estimate:  1.2E+017
```

The computed result is the correct solution (internally the toolbox routine makes use of the accurate dot product evaluation available in C-XSC).

5.4 Example: Cauchy principal value integral

The freely available package CLAVIS (**Cla**sses for **v**erified **I**ntegration over **S**ingularities) has been developed and implemented using C-XSC by Wedner as part of his thesis [20]. This package allows the computation of enclosures for definite integrals of several kinds (Riemann, Cauchy principal values, ...).
Let us start with two definitions:

The Cauchy principal value integral $I(f;\lambda)$ is defined as follows

$$I(f;\lambda) := \int\!\!\!\!\!\!-_a^b \frac{f(x)}{x-\lambda}\,dx := \lim_{\varepsilon\to 0+}\left(\int_a^{\lambda-\varepsilon}\frac{f(x)}{x-\lambda}\,dx + \int_{\lambda+\varepsilon}^b \frac{f(x)}{x-\lambda}\,dx\right), \quad \lambda\in(a,b)$$

and $f \in C^{2n+1}[a,b]$.

The nested integral $I(f;\lambda,\mu)$ is defined in the following way:

$$I(f;\lambda,\mu) = \int\!\!\!\!\!\!-_a^b \int\!\!\!\!\!\!-_c^d \frac{f(x,y)}{(x-\lambda)(y-\mu)}\,dy\,dx\,, \quad \lambda\in(a,b),\ \mu\in(c,d).$$

We now compute an enclosure of the nested integral

$$I(f;\lambda,\mu) = \int\!\!\!\!\!\!-_1^2 \int\!\!\!\!\!\!-_1^2 \frac{\sin(e^{x^2})\sin(e^{y^2})e^{x^2+y^2}}{(x-\lambda)(y-\mu)}\,dy\,dx$$

with $\lambda = 1.25$ and $\mu = 1.5$ using the CLAVIS library. The header file `"cubature.h"` belongs to the CLAVIS library. To be able to link the program `cubature.o` must be linked. The following program also demonstrates how exceptions may be handled.

```
#include <iostream>
#include "cubature.h" //don't forget to link cubature.o
//source code of this program is assumed to be in the clavis directory

using namespace std;
using namespace cxsc;
```

```
// cauchy x cauchy integral (using cauchy x cauchy formula)
//
// f(x,y) = sin(exp(y*y)) * exp(y*y) * sin(exp(x*x)) * exp(x*x)
//
// complete integrand of I(f; lambda, mu):  f(x,y) / ((x-lambda)*(y-mu))
//
// ------------------------------------------------------------------------

int main() {

   try {
      operand r( exp(sqr(y)) ), s( exp(sqr(x)) );
      integrand f = sin(r) * r * sin(s) * s;

      double lambda=1.25;  //singularity in x direction
      double mu=1.5;       //singularity in y direction
      double xlb=1, xub=2; //x-range of integration
      double ylb=1, yub=2; //y-range of integration
      double eps= 1e-6;    //required accuracy

      cauchy_integral example(f, lambda, mu);
      //compute an enclosure of I(f; lambda, mu):
      example.integrate(xlb, xub, ylb, yub, eps);
      cout << SetPrecision(8,2) << Scientific
              << "Required max. diameter of remainder:  " << eps << endl
              << SetPrecision(16,12) << example << endl;
   }//try
   catch(integrand::error e)
   { cout << " formelgen. " << e.i << endl; }

   return 0;
}//main

/* Output:

Required max. diameter of remainder:  1e-06
 number of intervals : 109 (44)
 #f                  : 17233

 approximationsum    : [-7.6237054671070354E+001,-7.6237054670795458E+001]
 d(approximationsum) : 2.7489477361086756E-010

 remainder           : [-4.9415981455851922E-007,4.9416704156171493E-007]
 d(remainder)        : 9.8832685612023414E-007

 enclosure           : [-7.6237055165230175E+001,-7.6237054176628404E+001]
 d(enclosure)        : 9.8860176933612820E-007

*/
```

The output shows, that

$$\int_1^2 \int_1^2 \frac{\sin(e^{x^2})\sin(e^{y^2})e^{x^2+y^2}}{(x-1.25)(y-1.5)}\,dy\,dx \in [-76.2370552, -76.2370541].$$

This result is guaranteed by the algorithm itself.

5.5 Example: Time measurements

We are frequently asked for timings. Here we give a frame for time measurements. The source code can be modified in an obvious way to do timings for other operations and functions.

```
//Simple frame for time measurements

#include <iostream>
#include <ctime>        //clock()
#include "interval.hpp" //interval operations
#include "imath.hpp"    //elementary functions for interval arguments

using namespace std;
using namespace cxsc;

void start_clock(clock_t& t1);  //function to start the timer
void print_time_used(clock_t t1);

int main()
{
   long iMax= 100000;
   cout << "Number of repetitions: "<< iMax << endl;
   interval x(200.0,200.001);
   clock_t t; //defined in <ctime>

   cout << "Elementary function calls ..." << endl;
   start_clock(t);
   for(long i=0; i<iMax;)
   {
      x= ln(exp(atan(sin(cos(x)))));
      i++; //avoid compiler optimization
   }
   print_time_used(t);
}

void start_clock(clock_t& t1)
{
   t1= clock();
   if (t1 == clock_t(-1))  //terminate if timer does not work properly
   {
      cerr << "Sorry, no clock\n";
```

```
      exit(1);
   }
}

void print_time_used(clock_t t1)
{
   clock_t t2= clock();
   if (t2 == clock_t(-1))
   {
     cerr<< "Sorry, clock overflow\n";
     exit(2);
   }
   cout << "Time used: " << 1000*double(t2-t1)/CLOCKS_PER_SEC
        << " msec" << endl;
}
/*
Results computed on a SUN Ultra 60 Workstation running Solaris 7
using GNU C++ Compiler Version 3.2 without any optimization:

Number of repetitions: 100000
Elementary function calls ...
Time used: 1370 msec
*/
```

Note that the given frame for time measurements is not so appropriate to measure very short or very long execution times.

6 Current Work on C-XSC

- Finish the final version C-XSC 2.0 (the actual version is: Betarelease 2 from December 2002)
- Modify the sources in such a way that C-XSC will run with more C++ compilers (e.g. with SUN Forte, Compaq, other compilers available for Windows systems; up to now C-XSC 2.0 only runs with GNU C++ compilers from version gcc 2.95.2 to version gcc 3.2.)
- Adaptation and completion of the C-XSC test suite to more C++ compiler versions (most C++ compiler do not conform completely to the C++ standard. This still causes problems when using the already existing rudimentary test suite. Meanwhile the installation of the C-XSC library is checked in the following way: Install also the numerical toolbox and look whether the toolbox programs deliver correct results. If the computed results are equal to the prestored correct values it is assumed that the C-XSC installation was successful.)
- Improve performance: due to the extensive use of the C++ exception handling, the extensive use of template classes, and the extensive use of function inlining it is (up to now) not possible to compile C-XSC with the GNU Compiler using e. g. the compiler option -O3 as optimization level

- For historical reasons C-XSC is build on emulations for several basic floating point operations. This makes the actual C-XSC run time system portable but slow compared to the speed of hardware operations. Nowadays most processors conform to the IEEE 754 standard. So, fast hardware operations are available for all rounding modes. These operations will be used in forthcoming C-XSC versions (at least for special processors like Intel and SUN)
- A thorough documentation of the routines available in C-XSC will be prepared. This is important because due to significant modifications concerning C++ most available documentation is no longer up to date
- Simplification and redesign of the runtime system (RTS). The RTS comprises rounding control, reliable input/output routines, routines to compute accurate dot products for data types real, complex, interval, and complex interval, ...
- Development and implementation of parallel versions of selfverifying solvers based on C-XSC and MPI on cluster computers

7 Acknowledgements

Many colleagues and scientists (see [4] Paragraph 1) have directly and indirectly contributed to the realization of C-XSC and C-XSC 2.0. The authors would like to thank each of them for his or her cooperation.

Thanks to the referees for valuable comments and suggestions.

References

1. Cuyt, A.; Verdonk, B.; Becuwe, S; Kuterna, P.: A Remarkable Example of Catastrophic Cancellation Unraveled. Computing 66, 309-320 (2001).
2. Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: C++ Toolbox for Verified Computing. Basic Numerical Problems. Springer-Verlag, Berlin (1995).
3. Hölbig, C.; Krämer, W.: Selfverifying Solvers for Dense Systems of Linear Equations Realized in C-XSC. Preprint BUW-WRSWT 2003/1, Universität Wuppertal (2003).
4. Hofschuster, W.; Krämer, W.; Wedner, S.; Wiethoff, A.: C-XSC 2.0: A C++ Class Library for Extended Scientific Computing, Preprint BUGHW-WRSWT 2001/1, University of Wuppertal, pp. 1-24 (2001).
5. ISO/IEC 14882: Standard for the C++ Programming Language (1998).
6. Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: C-XSC – A C++ Class Library for Scientific Computing. Springer-Verlag, Berlin (1993).
7. Krämer, W.; Bantle, A.: Automatic Forward Error Analysis for Floating Point Algorithms. Reliable Computing, Vol. 7, No. 4, pp 321-340 (2001).
8. Krämer, W.; Wedner, S.: Two adaptive Gauss-Legendre type algorithms for the verified computation of definite integrals. Reliable Computing Vol. 2, No. 3, pp. 241-253 (1996).
9. Krämer, W.; Kulisch, U., Lohner, R.: Numerical Toolbox for Verified Computing II. Advanced Numerical Problems. Draft version available: `http://www.uni-karlsruhe.de/~Rudolf.Lohner/papers/tb2.ps.gz`.

10. Kulisch, U.: The Fifth Floating-Point Operation for Top-Performance Computers or Accumulation of Floating-Point Numbers and Products in Fixed-Point Arithmetic. Bericht 4/1997 des Forschungsschwerpunkts Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, Universität Karlsruhe (1997).
11. Kulisch, U.: Advanced Arithmetic for the Digital Computer. Design of Arithmetic Units. Springer Verlag, Wien (2002).
12. Loh, Eugene and Walster, G. William: Rump's Example Revisited. Reliable Computing, Vol. 8, No. 3, pp. 245-248 (2002).
13. Lohner, R.: Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen. Dissertation, Universität Karlsruhe (1988).
14. Meyers, Scott: Effective C++, 50 Specific Ways to Improve Your Programs and Designs. Addision-Wesley (1998).
15. Meyers, Scott: More Effective C++, 35 New Ways to Improve Your Programs and Designs. Addision-Wesley (1997).
16. Neher, M.: Validated bounds for Taylor coefficients of analytic functions. Reliable Computing 7, pp. 307-319 (2001).
17. Rump, S. M.: Algorithms for verified inclusions – theory and practice. In: Moore, R. E. (ed.): Reliability in Computing, pp. 109-126, Academic Press, New York (1988).
18. Rump, S. M.: INTLAB - INTerval LABoratory. In Tibor Csendes (editor): Developments in Reliable Computing, pages 77-104. Kluwer Academic Publishers, Dordrecht (1999).
19. Stroustrup, B.: The C++ Programming Language. Special Edition, Addison-Wesley, Reading, Mass. (2000).
20. Wedner, S.: Verifizierte Bestimmung singulärer Integrale - Quadratur und Kubatur. Thesis, Univ. Karlsruhe (2000).